Introduction

These notes are an introduction to the basic organisation and functioning of a modern general-purpose digital computer. The main topics discussed are:

Data Representation: Introduces the different ways in which data may be represented in a computer system, and their basic operations.

Computer Organisation: Describes the main components of a modern computer system and their interrelationships.

The Motorola 68000 Chip: Describes the structure and operation of the 68K family of Motorola processors, and their addressing modes.

Digital Logic: Explains how logical circuits can be implemented with digital technology, including basic gates and their combinations.

Error Coding: Discusses the ways in which errors may be detected and corrected in transmitted information.

Assembly Language Programming: Presents an introduction to 68K assembly language programming, including branching, loops and parameter passing.

In order to give students as much time as possible to practice assembly language programming, these topics are not presented in this sequence in this course, but they are introduced as required to understand the concepts. Thus, expect to see a topic first, to be revisited later.

The diagram in Figure 1 depicts the organisation of the topics:

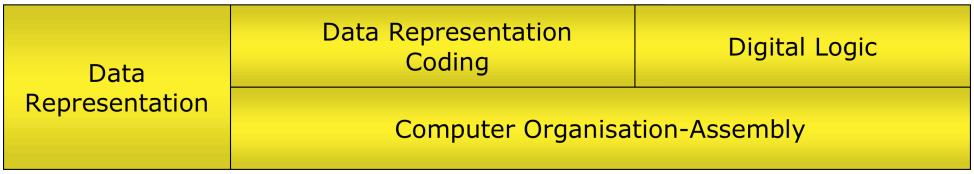

Figure 1: Course Organisation

Contents

List of Figures

List of Tables

List of Source Code Examples

Chapter 1

Data Representation

Objectives

After studying this chapter, students should be able to:

- Describe and use number systems in different bases
- Convert numbers between binary, decimal hexadecimal and octal
- Perform additions and subtractions in various numbers systems

1.1 Digital Information

1.1.1 Introduction

In modern times, information is very often represented and managed in a digital format. That is, in one way or another the information — photos, pictures, video, music, data — is translated to 0s and 1s, and manipulated that way. For example:

- a character can be represented A = 01000001, B = 01000010 ...
- colours on a gray scale can be represented by white = 000000 ... black = 111111
- yes/no or true/false can be represented by yes = 1, no = 0, etc.
- an open switch may be a 1, a closed switch may be a 0

Note that the meaning of the sequence of 0s and 1s is a matter for interpretation. For example, we'll see that the symbol 01000001 may be interpreted as the character A or

the number 65.

As a consequence of the success of the modern computer to be able to store and manage binary data, these representations are more common every day. In this course we are interested in the architecture and functioning of general purpose digital computers. From the software/programmer point of view, all information inside a computer are 0s and 1s. From the engineering point of view these are two voltages, 0 volts is interpreted as a 0, +5 volts(or possibly some lower voltage in newer systems) is interpreted as a 1.

A computer spends much of its time performing arithmetic operations. Our arithmetic is based on the "positional system" where the position of a digit within a number determines its actual value; the number 2 in 123 means twenty, but the number 2 in 2,345 means two thousands. The positional value of each column, starting from the right is 1, 10, 100, 1000, etc. That is $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, $10^3 = 1000$, etc. The decimal number system has a Base or Radix of ten, using ten different digits or symbols in the system 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. All integers are made up of a combination of these digits, in the positional system described before.

Although we are used to the decimal system (i.e. to the base 10) number systems can be based on any radix such as 2, 5, 8, 10, 16, 28, 112, etc. The arithmetic rules in any system are similar. Although it may seem surprising at first, the addition and multiplication rules are quite the same, but the tables change. For example, if we work to the base 5, we count:

1, 2, 3, 4, 10, 11, 12, 13, 14, 20, ...

So the addition rules look different to the decimal ones, as in:
$1 + 2 = 3$
$2 + 3 = 10$
$3 + 3 = 11$
$4 + 3 = 12$

And the times tables also change:

2 * 1 = 2

2 * 2 = 4

2 * 3 = 11

2 * 4 = 13

What would be the 3 times table to the base 5? (Complete Table 1.1)

Table 1.1: 3 multiplication table, to the base 5

Operation	Result
3 * 1	3
3 * 2	
3 * 3	
3 * 4	

We can also perform other operations, quite similar to the standard decimal ones. As an exercise, complete the operations in Table 1.2 (only one has been completed).

Table 1.2: Some operations to the base 5

Operation	Result
3^2	14
2^3	
4^2	
3^3	

1.2 Number Representation

Since inside a computer everything is represented with 0s and 1s, the most important number systems for digital computers are:

- binary: base 2. Digits are: 0,1
- octal: base 8, Digits are: 0, 1, 2, 3, 4, 5, 6, 7

- hexadecimal: base 16. Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

We shall see that hexadecimal and octal numbers are closely related to binary numbers, and we'll learn to use them and do arithmetic in each of them.

1.2.1 Binary Numbers

- The basis for binary numbers is 2
- Radix = 2
- The symbols are: 0,1

Similarly to decimals, the positional value of each binary digit indicates the corresponding power of 2, as follows:

Table 1.3: A binary number

digit	1	1	0	0	1	1	0	1
power	128	64	32	16	8	4	2	1
exponent	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Thus, according to Table 1.3 the number 11001101 is equivalent to the decimal:
$1*2^7+1*2^6+0*2^5+0*2^4+1*2^3+1*2^2+0*2^1+1*2^0 = 128+64+8+4+1 = 205$

NOTE: We are calculating the expression above using *decimal* operations.
Exercise: find the decimal equivalent of the binary numbers:
1101 =
1111111 =
10000000 =

Notation: a binary digit is called a bit.

Some digits have meaning for different bases, such as 10010 which may be a binary, octal, decimal or hexadecimal number. Hence, it is often necessary to indicate the basis, such as in 10010_2, 74562_8 and 74562_{10}.

Sometimes a prefix is used, as in % for binary and $ for hexadecimal.

Counting in Binary

Table 1.4: Counting in Binary

Decimal	Binary	Decimal	Binary
00	00000	11	
01	00001	12	
02	00010	13	
03	00011	14	
04		15	
05		16	
06		17	
07		18	
08		19	
09		20	
10		21	

Exercise: complete the table.

Use the prefix % or suffix subscript 2 eg = %110110 or 11110010_2

1.2.2 Hexadecimal Numbers

Although binary numbers are very easy for computers to store and manipulate, for human beings they are quite uncomfortable to interpret and use. For example, the number 10101111001100111111000000101 would be very hard to remember for a human being. We shall see that its equivalent in hexadecimal is AF33D05, which is much easier. Both hexadecimal and octal are important because the conversions between both systems and binary are quite straightforward.

- The basis for hexadecimal numbers is 16.
- Radix = 16 (2^4)
- The symbols are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Table 1.5: Hexadecimal Numbers

Positional values:	16^4	16^3	16^2	16^1	16^0
Decimal values:	65,536	4096	256	16	1

Therefore the number:

$1D7E_{16} = 1 * 16^3 + D * 16^2 + 7 * 16^1 + E * 16^0$

$= 1 * 4096 + 13 * 256 + 7 * 16 + 14 * 1 = 7550_{10}$

Exercises: Convert to decimal

$7E =

$A8 =

$13BF =

Counting in Hexadecimal

Table 1.6: Counting in Hexadecimal

Decimal	Hexadecimal	Decimal	Hexadecimal
00	00	11	
01	01	12	
02	02	13	
03	03	14	
04		15	
05		16	
06		17	
07		18	
08		19	
09		20	
10		21	

Exercise: complete Table 1.6.

For hexadecimal numbers use prefix $ or subscript 16, as in $1FED or $1FED_{16}$

1.2.3 Octal Numbers

- The basis for octal numbers is 8.
- Radix = 8 (2^3)
- The symbols are: 0, 1, 2, 3, 4, 5, 6, 7

Table 1.7: Octal Numbers

Positional values:	8^4	8^3	8^2	8^1	8^0
Decimal values:	4096	512	64	8	1

$1573_8 = 1 * 8^3 + 5 * 8^2 + 7 * 8^1 + 3 * 8^0 = 1 * 512 + 5 * 64 + 7 * 8 + 3 * 1 = 512 + 320 + 56 + 3 = 891_{10}$

Counting in Octal

Table 1.8: Counting in Octal

Decimal	Octal	Decimal	Octal
00	000	11	
01	001	12	
02	002	13	
03	003	14	
04		15	
05		16	
06		17	
07		18	
08		19	
09		20	
10		21	

Exercise: complete the table.

1.3 Conversions Between Number Systems

1.3.1 Binary to Decimal

Since we know how to operate in decimal, using the positional value of each digit, we can calculate the decimal expression of a binary number:

Table 1.9: Binary to Decimal

2^4	2^3	2^2	2^1	2^0
16_{10}	8_{10}	4_{10}	2_{10}	1_{10}

For example:

$10101_2 = 1 * 16 + 0 * 8 + 1 * 4 + 0 * 2 + 1 * 1 = 16 + 0 + 4 + 0 + 1 = 21_{10}$

Exercises:

Convert the following binary values to decimal:

```
10101 =

01110 =

11110 =

01101110 =

11100111 =
```

1.3.2 Decimal to Binary

Repeatedly divide by 2; for example, convert 28_{10} to binary

2)	28	
2)	14	0
2)	7	0
2)	3	1
2)	1	1
2)	0	1

In the algorithm above, the resulting binary number should be read upside down. Therefore: $28_{10} = 11100_2$

The above algorithm can be explained as follows. According to the above sequence, we can write:

28 = 14 * 2 + 0

14 = 7 * 2 + 0

7 = 3 * 2 + 1

3 = 1 * 2 + 1

1 = 0 * 2 + 1

Then,

28_{10} = 14 * 2 + 0 = (7 * 2 + 0) * 2 + 0 = 7 * 2^2 + 0 * 2^1 + 0 * 2^0 =

= (3 * 2 + 1) * 2^2 + 0 * 2^1 + 0 * 2^0 = 3 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 =

= (1* 2 + 1) * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 =

= 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 11100_2

Exercises:

1. Convert the following decimal numbers to binary:

   8

   17

   32

   64

   127

   128

   255

2. What is the largest number that may be represented using 4 bits? 8 bits? 16 bits? and 32 bits?

3. How many bits do you need to represent 1000_{10}? and 100000_{10}? and 100000000_{10}?

1.3.3 Hexadecimal to Binary

The main advantages of using hexadecimals is that they are convenient for human beings to use, and that it is very easy to convert between them and binary. To convert hexadecimal to binary, write each hexadecimal digit as a 4-bit binary number and put it all together as in the following example:

Table 1.10: Hexadecimal to Binary

Hexadecimal	F	D	6	9	A
Binary	1111	1101	0110	1001	1010

The binary expression of $FD69A is %11111101011010011010. In particular, since a byte is 8 bits, each memory location may be represented exactly by 2 hexadecimal digits.

Exercises:

Convert the following hexadecimal values to binary:

```
$FE45   =
$BBB775 =
$ABCDEF =
```

1.3.4 Binary to Hexadecimal

Conversely, to convert from binary to hexadecimal we break down the binary number into groups of 4 binary bits (padding with 0s to the left if necessary) to get the equivalent hexadecimal expression, as in:

Table 1.11: Binary to Hexadecimal

Binary	0101	1110	1010	1011	1111	0001
Hexadecimal	5	E	A	B	F	1

NOTE: Start at the right hand end and fill the left hand end with zeros to get 4 bits if necessary.

Exercises:

Convert the following binary values to hexadecimal:

```
%1 0000 1010 1110 1111 0010 =
%0001 0001 0011 1010 1011 =
%10100011110001100011000 =
%11111100010101010010 =
```

1.3.5 Conversions Between Octal and Binary

These conversion are the same as the hexadecimal conversions, but instead of dividing into groups of 4 you have to divide into groups of 3.

1.4 Operations with Numbers

1.4.1 Decimal Addition

When we learn to add in decimal, we learn some rules that are appropriate for that number system, such as $2 + 3 = 5$ and $7 + 6 = 13$. Using these rules we have learnt how to add multi-digit numbers such as:

$$\begin{array}{rr} & 56 \\ + & 97 \\ \hline & 153 \end{array}$$

When we add $6 + 7$ we obtain 13, so we write down a sum of 3 and a carry of 10 to the next column:

$56 + 97 = 50 + 6 + 90 + 7 = 50 + 90 + 6 + 7 = 50 + 90 + 13 =$

$= 50 + 90 + 10 + 3 = 150 + 3 = 100 + 50 + 3 = 153$

1.4.2 Binary Addition

The rules to add binary numbers are the same as for decimals, but remembering that $1 + 1 = 10$. This, of course, means that when adding $1 + 1$ you have to write down 0 and carry 1. The following table summarises the rules for binary addition:

Table 1.12: Binary Addition

	Sum	Carry
$0 + 0 =$	0	0
$0 + 1 =$	1	0
$1 + 0 =$	1	0
$1 + 1 =$	0	1

Example:

$$\begin{array}{cccccc} & 1 & 0 & 1 & 0 & = 10_{10} \\ + & 0_0 & 0_1 & 1_0 & 1 & = 03_{10} \\ \hline 0 & 1 & 1 & 0 & 1 & = 13_{10} \end{array}$$

Exercises:

Perform the following binary additions:

```
1010 + 0011 =

1110 + 0001 =

1101 + 0001 =

1101 + 1001 =

1110 + 1101 =

0101 + 1000 =

0001 + 1001 =
```

1.4.3 Computer Data Sizes

Although the information to be represented in a computer usually has no fixed limit, a limitation of computers is that they deal with data in a fixed number of bits, such as 8, 16, 32, 64 or 128 bits. This means that the result of an arithmetic operation might exceed the size currently being used by the computer. If the result is too big an error will occur, and it has to be managed properly. For example, in 8 bits the following operation produces a 9^{nth} bit:

$$10011101 + 10001001 = \mathbf{1}00100110$$

The leftmost bit (in bold) of the result is a *carry out* bit. Although the original operands are 8 bits each, the result of the operation does not fit into 8 bits. If we don't consider the carry out bit, this operation is (in decimal)

$$157 + 137 = 38$$

which is clearly wrong. The carry out bit is telling us that the result exceeds the 8 bits capacity, and that the result is not valid.

RULE: For unsigned data (more on this later), if the carry out bit is 1 the result is invalid.

Exercises:

Perform the following unsigned binary additions and determine the validity of the result:

```
0111 0001 + 0100 0001 =

1110 1110 + 1111 1101 =

0101 1101 + 1010 0101 =
```

1.4.4 Binary Subtraction

Compare the following two operations:

7	73
– 4	–17
———	———
+3	56

Whereas the first operation on the left hand side presents no problems, on the right hand column of the second example we have 3–7 which we can't do. So, we add the base 10 to the 3 to give 13 producing a borrow of one to the next column (that is, an actual borrow of 10), and then subtract the 7 to give a difference of 6. What we are actually doing is:

$$\begin{array}{rcrcrl} 73 & = & 60 & + & 13 & \\ & & -10 & - & 7 & = \\ & = & 50 & + & 6 & = 56 \end{array}$$

Similarly, we can subtract binary numbers — remembering that all the operations are binary — like so:

$$\begin{array}{rcccccc} & & 1 & & & & \\ & 1 & 0 & 1 & 1 & = & 11_{10} \\ - & 0_1 & 1 & 1 & 0 & = & 6_{10} \\ \hline & 0 & 1 & 0 & 1 & = & 5_{10} \end{array}$$

Table 1.13 summarises the rules for binary subtraction.

Table 1.13: Binary Subtraction

	Difference	Borrow
0 - 0 =	0	0
0 - 1 =	1	1
1 - 0 =	1	0
1 - 1 =	0	0

NOTE:

- For unsigned numbers, the top number must be larger than the bottom number. It is not possible to subtract a larger number from a smaller one, since this will produce a borrow *and the result will be invalid.*
- In the figure the need to borrow is indicated by the single 1 at the top, and the little 1 on the right-hand side of the last column.

Exercises

Without doing the calculations, determine whether the following subtractions of unsigned binary data are valid. Then, perform the calculations and confirm your conclusions. Remember that if there is a borrow out of the last column, the result is invalid.

```
0101 1110       1100 1111    0011 1111
      - 0011 0110     - 1001 0001 - 1000 0110
```

1.4.5 Hexadecimal Addition

By memorising the value of the addition of hexadecimal digits, we can add hexadecimals. Similarly to base 10, this is the "add tables" in the base 16. Of course, we can always use decimal, but this is not really necessary. Table 1.14 shows some hexadecimal additions.

Table 1.14: Hexadecimal addition table

3 + 5 = 8	5 + 5 = A
3 + 6 = 9	5 + 8 = D
3 + 7 = A	7 + 8 =
3 + 8 = B	8 + 8 =
A + 3 =	A + 4 =
A + 5 =	A + 6 =
B + 7 =	D + 8 =

Exercise: complete Table 1.14.

Example:

$$\begin{array}{rr} & \$1F4C3 \\ + & \$729AB \\ \hline & \$91E6E \end{array}$$

3_{16} +B_{16} = E_{16} and carry = 0

C_{16} +A_{16} = 16_{16} sum 6_{16} and carry = 1

We can also construct multiplication (times) tables.

Table 1.15: Some hex operations

2 * 3 = 6	2 * 8 = 10	2 * D =
2 * 4 = 8	2 * 9 = 12	2 * E =
2 * 5 = A	2 * A = 14	2 * F =
2 * 6 = C	2 * B =	3 * 8 =
2 * 7 = E	2 * C =	3 * A =

Student Exercises:

- Perform the following hexadecimal additions. Check your results for validity.

```
  $1EDF      $57DA    $FC45
+ $7EC0    + $7BCF  + $DEA9
  -----          -------       -------
```

- Write down the complete hexadecimal 3-multiplication table.

1.4.6 Hexadecimal Subtraction

By remembering the addition rules of hexadecimal digits, subtraction is straightforward. Of course, we can always use decimal, but this again is not really necessary. Consider the following example:

	$\$F_1$	5_1	B
–	\$3	A	9
	\$B	B	2

When the top hex digit is too small, we do the same as in decimal arithmetic. For example, for the final column we have:

B_{16} -9_{16} = 2_{16} and borrow = 0
5_{16} - A_{16} = B_{16} and borrow = 1
F_{16} - 3_{16} - 1_{16} (the borrow) = B_{16}

Student Exercises: First assess the validity of the following subtractions on hexadecimal unsigned numbers. Then perform the operations and confirm your results.

```
  $F562  $BACD      $BEAD $DE35    $4567
  $987A
- $C453    - $2FFE    - $00F5    - $F17D  - $DEED
- $AFF3
```

1.5 Range of Positive Binary Numbers

Table 1.16: N^o of bits vs. Range

Bits	Range/Max Value	N^o of values
8	0 - 255	256
16	0 - 65,335	65,536
32	0 - 4,294,967,295	2^{32}-1
64	0 -18,446,744,073,709,551,615	2^{64}-1

The maximum value is 2^n-1, i.e. for 8 bits it is 2^8-1 = 255

Practical Work 1

1.6 Tutorial exercises

1(a) Build the complete times tables in base 5. For example, 4_5 * 1_5 = 4_5, 4_5 * 2_5 = 13_5, 4_5 * 3_5 = 22_5 ...

(b) Convert the decimal number 173 to the base 3 and to the base 5

2. Find all the powers of 4 to the base 5

3. What is the largest number that you can get with 4 bits, with 8 bits and 16 bits?

4. Convert the following decimal numbers to base 7:

(a) `21`

(b) `59`

(c) `98`

5. Convert the following decimal numbers to *binary*, *octal* and *hexadecimal*:

(a) `27`

(b) `3`

(c) `114`

(d) `55`

(e) `32`

6. Perform the following conversions:

(a) Binary to decimal:

```
01111111 =
11111111 =
```

(b) Binary to hexadecimal:

```
%1 1100 1010 1110 1111 1111 =
%0001 1111 0011 0101 1011  =
```

`%10100011100001100011000 =`

`%11111100010101010010 =`

(c) Hexadecimal to binary:

`$FE45 =`

`$BBB775 =`

`$ABCDEF =`

1.7 Laboratory exercises

1. Log on to the Learning Hub:

   `www.rmit.edu.au/online`

   Ensure that you can access both Blackboard and WebLearn interfaces for this course. If you don't have both of these entries for `COSC1082`, email the head tutor <`robert.mcquillan@rmit.edu.au`> with your student number and access will be arranged.

2. Browse the sections available on the course's Blackboard. The Official Announcements forum will be used to communicate important course-related messages such as changes to assessment, so at the very least you should inspect this one forum regularly. All students are encouraged to participate in the discussion forums.

3. Connect to WebLearn and complete Quiz 1. You should do this at least 5 times so that you get exposed to most of the questions in the quiz bank. (A different version of the quiz will be generated each time.) Discuss your answers with your lab assistant if needed, or alternatively open a discussion on the Blackboard.

4. Use any remaining time to seek assignment-related help or ask other course-related questions.

Chapter 2

Data Representation (cont.)

Objectives

After studying this chapter, students should be able to:

- Describe how negative numbers may be represented in a computer system
- Describe the 2's complement number system
- Perform operations that include negative numbers using the 2's complement representation
- Describe the main components and operation of a general purpose computer and their operation, including CPU, memory and buses

2.1 Representing Negative Values

We have been able to represent positive integer numbers using 0s and 1s. The two most common systems to represent negative numbers are:

- Signed Magnitude
- Two's Complement

2.1.1 Signed Magnitude

The most significant bit is used to indicate the sign, using 1 = negative and 0 = positive. For example, binary 0000 0110 is used to represent +6, and binary 1000 0110 is used to represent –6. This representation is clear and easy to implement. However, the operations do not conform to the standard arithmetic rules, as in:

$$\begin{array}{rll} & 0000\ 0110 & = +\ 6 \\ + & 1000\ 0110 & = -\ 6 \\ \hline & 1000\ 1100 & = -\ 12 \end{array}$$

This is bad for circuitry, because all the circuits are designed to implement the rules of standard arithmetic, and this approach will mean that special circuits would have to be designed and built for this representation.

2.1.2 Representing Negative Binary Values: 2's Complement

The problem with signed magnitude is that the choice of negatives is a poor one from the point of view of the operations. We need *a different way* to select them so half of the numbers are positive and the other half are negative. To this end, we choose the negatives starting from -1 at the other end of the interval, as shown in the following diagram for 8 bits:

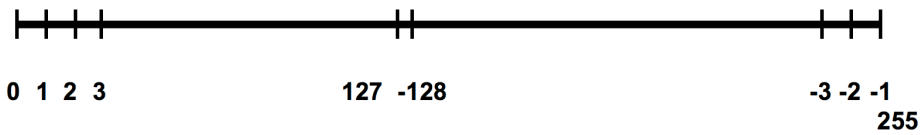

Figure 2.1: 2's complement representation

With this representation, when we add in binary two opposite numbers we get the result `1 0000 0000` (i.e. 256), so if we ignore the carry out we get all zeros, which is the result we want. The carry is not in the resulting byte anyway, so it is easy to ignore. We can see that this is more convenient from the point of view of the operations:

$$\begin{array}{rr} 1 = & 0000\ 0001 \\ -1 = & 1111\ 1111 \\ \hline & 1\ 0000\ 0000 \end{array}$$

Summarising, for 8 bits we have:

- unsigned numbers: no negatives, 0 to 255
- 2's complement: 0 (0000 0000) to 127 (0111 1111) positive, –128 (1000 0000) to –1 (1111 1111) negative.

Thus, if we ignore the carry out we can use the existing electronic circuits for these operations. For example, in the 68K processor the operation

$$5 + (-5) = 00000101 + (-11111011) = 100000000$$

is performed, the `0000 0000` byte is left in a register and two special bits, the *carry bit C* and the *extend bit X* are set to 1, because there has been a carry out. When needed, programs can check these bits to find out whether there was a carry out of the last column. If we only consider 8 bits the operation is correct.

Positional in 2's complement

There is an alternative very useful way of looking at the 2's complement representation. For an 8-bit number we consider the positional values as follows:

Binary	-2^7	$+2^6$	$+2^5$	$+2^4$	$+2^3$	$+2^2$	$+2^1$	$+2^0$
Decimal	– 128	64	32	16	8	4	2	1

In this representation, the number:
$1111\ 1111 = -1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 =$
$-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -1$

NOTE:

- The most significant bit has a negative contribution, –128 in this case.
- In 8 bits the binary sum of a number and its 2's complement opposite is: 1 0000 0000 = 256 = 0

Student Exercises

Find the 8-bit 2's complement expression of the following decimals:

`+47 =`

`+63 =`

`-123 =`

`-77 =`

`-96 =`

NOTE: positive values are left unchanged.

2.2 Operations with Negative Values

When operating with negative values, we have to be careful how we interpret the operands and the results of the operations. For example, the following sum:

$$01000110 + 10101100 = 11110010$$

may be interpreted as an unsigned operation, or as a signed 2's complement operation:

$$\begin{array}{rrcrcr} & 0100\ 0110 & = & 70 & = & 70 \\ + & 1010\ 1100 & = & 172 & = & -84 \\ \hline & 1111\ 0010 & = & 242 & = & -14 \end{array}$$

NOTE: These operations are *the same*, only the *interpretations* are different. Hence, both types of operations may be implemented with the same circuits. Further, the 68K processor will simply perform the operation and set or reset some bits according to the result. These are the bits discussed in section 2.3.

2.2.1 Conversion of binary values to the 2's complement format

Let us consider 8-bit binary numbers. The rules for conversions are simple, based on flipping the bits of the binary number by changing 0s to 1s and 1s to 0s — this is called the *1's complement* — and adding binary 1 to obtain the 2's complement expression of a number `x`. The complete algorithm is as follows:

```
if 0 <= x <128
    the 2's complement expression is the same as the
    original number
else
    1. Find the 1's complement of x by flipping the bits
    2. Add 1
    3. The result is the 2's complement expression of x
```

Example: find the 2's complement expression of -35_{10}

First, 35_{10} =0010 0011_2

```
1. flip each of the bits above  1101 1100
2. add 1        + 0000 0001
3. result  1101 1101
```

This is the 2's complement expression of the number -35_{10}, which is the opposite of 35_{10}.

2.2.2 2's Complement Addition

2's complement addition is similar to unsigned binary addition, in that there are invalid results as a result of the sum falling outside the acceptable range of numbers. This cannot happen when you add two numbers with different sign, one positive and one negative. The addition *may be* invalid only when we add two positives or two negatives. The addition is invalid when the sum of two positives gives a negative result, or when the addition of two negatives gives a positive result.

We saw that in unsigned addition an invalid result was detected by a carry out of the MSB, that is, when the last column produces a carry out bit 1 (not to be confused with the carry bit C of the 68K processor). If the carry out bit is 1 the result is invalid *for unsigned data*. In *unsigned subtraction* we also saw that if there is a borrow out of the last column that result is invalid.

In 2's complement operations, an operation is not valid when there has been *overflow*. This is detected by the *overflow bit V*, so when V = 1 the result is invalid. To calculate the overflow bit V we (and the 68K processor) look at the carry into the last column (that is, the MSB or the sign bit), and the carry out of the last column. If both carry bits are the same (both 0 or both 1), the overflow bit V = 0; if they are different, the overflow bit V = 1. In the latter case, overflow has occurred and the operation is invalid.

Examples:

1. Consider:

$$\begin{array}{cccccccccc} & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ + & 1_0 & 1_0 & 1_0 & 0_1 & 1_0 & 1_0 & 0_0 & 0 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{array}$$

   In this example, the carry into the last column is 0, and the carry out is 1; therefore the carries are different, V = 1 and the result is invalid. Please observe that in the example the sum of two negative numbers produces a positive result, which is clearly wrong.

2. If we consider a number and its opposite, as in:

$$\begin{array}{cccccccccc} & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

   This example was the motivation for our discussion of 2's complement. Please observe that both the carry in and carry out are 1 (V = 0), so there is no overflow and

the result is correct as we wanted.

2.3 Control Bits

When a processor such as the 68K performs an addition or a subtraction, it typically interprets the operands as signed operands and performs the operation. At the end of it, it sets/resets some bits according to the result. For example, in the Motorola 68K:

- if an addition produces a carry out, the carry C and the extend X bits are set (=1), otherwise they are reset (=0);
- if a subtraction produces a borrow, the carry C and the extend X bits are set (=1), otherwise they are reset (=0), as above;

In both cases:

- if the result is negative, the negative bit N is set, otherwise it is reset;
- if the result is zero, the zero bit Z is set otherwise it is reset;
- if the operation results in overflow, the overflow bit V is set, otherwise it is reset.

This allows the programmer to determine exactly the result of the operation, depending on the interpretation that he/she wants to give.

1. Following our previous example, the sum:

$$\begin{array}{rr} & 0100\ 0110 \\ + & 1010\ 1100 \\ \hline & 1111\ 0010 \end{array}$$

results on both the C, X and V bits to be reset to zero, so the operation is valid both as a signed and unsigned operation. The result may be interpreted validly either as the sum of the two unsigned numbers, or as the sum of two signed numbers.

2. Let's now consider the operation:

$$\begin{array}{cc} & 1100\ 0110 \\ + & 1010\ 1100 \\ \hline 1 & 0111\ 0010 \end{array}$$

This results in C = 1, X = 1, and V = 1 (since the carry in is zero and the carry out is 1), and so both results are invalid.

3. Finally, consider:

$$\begin{array}{cc} & 1100\ 0101 \\ + & 0011\ 1011 \\ \hline 1 & 0000\ 0000 \end{array}$$

In this operation there is a carry out (C = 1 and X = 1), but there is also a carry in, so V = 1. Therefore, the operation is invalid as an unsigned operation, but it is valid as a signed operation.

Exercise:

Show an example of an operation in which C = 0 and V = 1.

2.4 Subtraction of 2's Complement Binary Numbers

Since we know how to add with negative numbers, we can subtract a number from another and get a correct result, provided the numbers are within the 2's complement range. The operation is performed by calculating the opposite in the 2's complement notation and adding. For example, if we want to do 0001 1110 – 0110 1100 we first have to find the expression of the opposite of the subtrahend 0110 1100:

flip the bits		1001 0011
add 1	+	0000 0001
result		1001 0100

Now add the number and the opposite just calculated:

	0001	1110
+	1001	0100
	1011	0010

Please note that the result is the number -78_{10}, that is, a negative result. The result is valid because it is the sum of a positive number and a negative number.

While this method works for many cases and is often simpler to implement, it has two shortcommings:

1. It simply will not work if the negative value is negative 128. This should be intuitively obvious since the range of values represented in 2's complement are $0 \rightarrow 127$ in the positive range and $-1 \rightarrow -128$ in the negative range. Clearly the complement of -128 should be 128 which is out of range.

2. Also, adding a 2's complement has the effect of moving some of the carries in the addition as compared to what would occur with a subtraction – this means that the C and V bits are not necessarily correct at the conclusion of the calculation.
   So, if knowing the value of these bits is critical, the only option is to subtract in the standard manner.

It is *very important* to understand that this approach to subtracting two 2's complement binary numbers *cannot* be used for *unsigned* binary numbers. To perform a subtraction on two unsigned binary numbers, it is *necessary* to perform the subtraction as you would for two decimal numbers. In summary:

- if the numbers are unsigned, perform the subtraction as for two decimal numbers
- if the numbers are signed, add the opposite of the subtrahend

For each binary number in the following subtraction, we may assume it may represent either a 2's complement value or an unsigned value:

binary	2's comp	unsigned
11011110	-34_{10}	222_{10}
–11010011	$-(-45)_{10}$	-211_{1}
00001011	$+11_{10}$	11_{10}

The computer performs the operation as a signed operation, so it is up to the user to interpret the operation by checking the C or V bits. In the example above C = 0, X = 0 and V = 0, so the results are valid in all interpretations.

Exercise:

Perform the following operations:

```
  1111 0000       0000 0001       1010 1011
+ 0111 1101     - 1111 1111     - 1111 1110


  0111 1111       1110 1111   1010 1111
- 1111 1111     + 0111 1110     - 0111 0001
```

Determine the V and C bits for each example and determine whether the operations as signed and unsigned numbers are valid.

Common Confusion

A common confusion arises when talking about the 2's complement format. Consider carefully the following statements:

1. Find the 2's complement expression of -35_{10}: this is $1101\ 1101_2$, as we have seen before.
2. Find the number that is the opposite of 35_{10} in the 2's complement format: this is -35_{10}, with the expression as above.
3. Given a number x, –x is the number that is the opposite of x in the 2's complement format, e.g. -35_{10} is the 2's complement of 35_{10}, and 18_{10} is the 2's complement of -18_{10}.
4. When we are talking of operations in the 2's complement format, to calculate x – y we should:
   (a) calculate the number –y (opposite of y in the 2's complement format), and
   (b) add this to x

Exercises

Find the 2's complement expression of the following numbers:

–18 =

+17 =

–63 =

+64 =

–127 =

+127 =

–128 =

+128 =

–1 =

–2 =

2.5 Basic Computer Organisation and Operation

Computers work by executing programs. A computer program consists of a set of instructions that the computer executes, e.g. `add`, `subtract`, `move` (to/from) memory, etc. The CPU (Central Processing Unit) operates by fetching and decoding each instruction, and then executing the instruction. To this end, the CPU contains an *Arithmetic and Logic Unit (ALU)* and a few locations with a fixed number of bits — typically 16, 32 or 64 bits — called *registers* to store binary information for immediate manipulation. Thus, the operation of a computer is based on the transfer of data from memory into the CPU area — ALU and registers — where arithmetic and logical operations are performed, and then writing the results back to memory. Figure 2.2 depicts a simplified computer configuration.

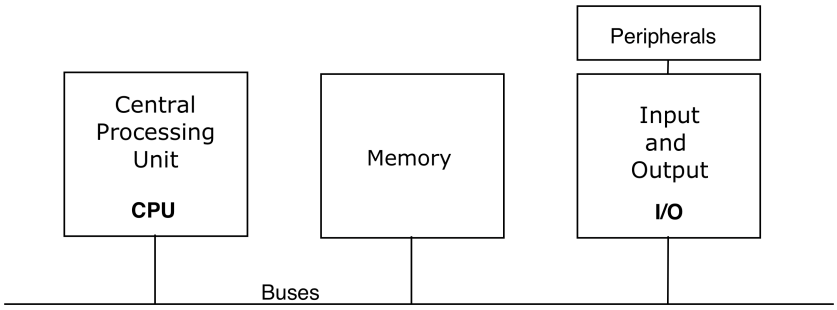

Figure 2.2: A computer block diagram

The main components of a computer are:

Memory: stores the data and instructions of currently running programs. (There may be more than one running program, each is called a *process*.) Main memory stores instructions and data in locations numbered sequentially: 0, 1, 2, 3, 4, 5, ... Each memory location is an *address*. Thus, an address is a location in memory that may be accessed by programs. Individual bits are *not addressable*; typically bytes are the smallest unit to have an address. This is expressed by saying that the memory is *byte addressable*. In each address programs can find data—the contents of the address—which should not be confused with the address itself. (See Figure 2.3).

Since N bits can express a binary number of size up to $2^N - 1$, the size of addressable

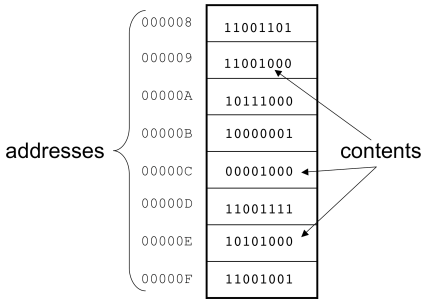

Figure 2.3: Memory locations and contents

memory is limited by the number of bits used to express an address. With N bits it is possible to have addresses ranging between 0 and $2^N - 1$, a total of 2^N addresses. The largest memory address with 16 bits is $2^{16} - 1$, and with 32 bits $2^{32} - 1$

CPU: fetches decodes and executes instructions stored in memory. It includes the ALU and registers described before, and other spaces such as the MAR (Memory Address Register), MBR (Memory Buffer Register) and PC (Program Counter) described in Chapter 3.

Input/Output System (I/O): connects the computer to external peripherals such as disks, keyboard, monitor, mouse, printer and network. The CPU communicates with some interface registers to handle the data transfers. This may be implemented by making the I/O addresses part of the memory, *memory-mapped I/O*, or by using special instructions to access special I/O registers, *instruction-based I/O* (also called *programming-based I/O*).

Bus(es): a bus is a collection of wires carrying binary information between subsystems: the CPU, registers, memory and I/O devices. A bus is normally shared between several devices, so there must be some sort of agreement on how to use the bus : *bus protocol*. A bus may be connecting two specific devices, *point-to-point bus*, or it can be a common pathway for several devices, called a *multi-point bus*.

Computer clock: The operations of a computer are synchronised by a computer clock, a hardware device that produces a very regular sequence of 'ticks'. Modern desktop computers clocks can tick in excess of 3 billion times per second, and

that gives an indication of the speed of the computer. In general, large computers and super computers are much faster than personal computers.

Usually, programmers write programs in a high level language such as Java or C, and then a compiler translates this to machine instructions. In doing so, the compiler works with the operating system to allocate memory for the program to run, including program instructions, room for global data variables and for the temporary storage of data such as function parameters and return addresses (more on this later). Although this is highly dependent on the system, Figure 2.4 shows a typical allocation of memory for a running program (more on this later).

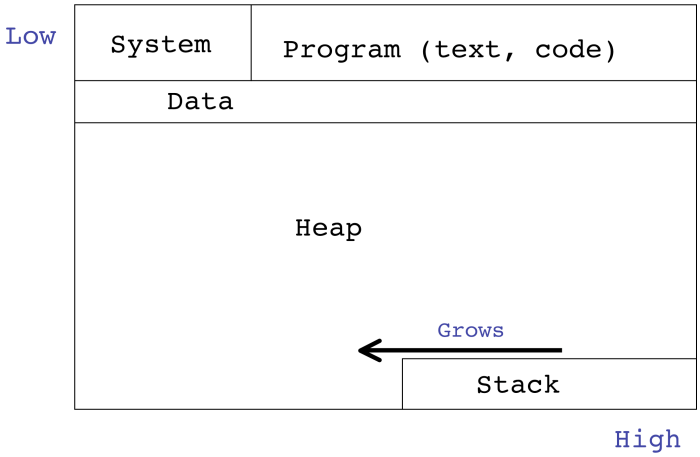

Figure 2.4: Typical memory configuration

2.5.1 Basic Operation

The number of parallel wires in the bus is called the *bus width*, and corresponds to the number of bits allowed at one time on the bus. A bus would have some lines dedicated to transfer data (*data bus*), some lines dedicated to handle addresses (*address bus*) and some control lines (*control bus*). When fetching data in, the CPU puts the memory address to be accessed on the address bus, sets the Read/Write line to read, and when the data is located in memory, it is transferred on the data bus usually into the CPU

area. When placing data in memory, the CPU places the destination location on the address bus, sets the Read/Write line to write, and puts the data on the data bus for the data to be placed in memory.

Usually, data and instructions are loaded into some of the CPU registers, operated on, and saved back to registers or memory. When processing an instruction, it is typical of computers to transfer at once quantities greater than a byte, such as words (2 bytes in the 68K) or long words (4 bytes in the 68K). This decision depends on several factors, such as the type of instruction being executed, the size of the data to be transferred, and the width of the bus. Figure 2.5 illustrates the interaction between subsystems and the data and address buses.

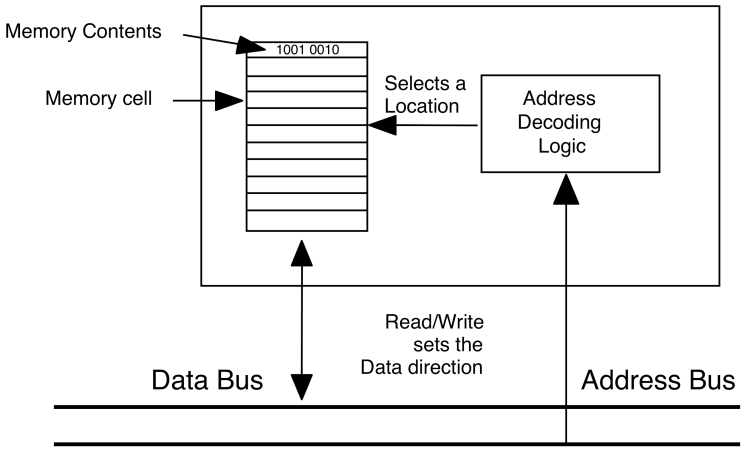

Figure 2.5: Memory organisation

Practical Work 2

2.6 Tutorial exercises

1. Perform the following unsigned binary additions and determine the validity of the result for 8 bits:

```
0111 1001 + 0100 0001 =
1110 1110 + 1110 1101 =
0111 1101 + 1010 0101 =
```

2. Perform the following unsigned binary subtractions and determine the validity of the result:

```
  0101 1110        1100 1111        0011 1111
- 0011 0110      - 1101 0001      - 1000 0110
```

3. Perform the following hexadecimal additions and determine the validity of the result:

```
  $1EDF          $57DA          $FC45
+ $7EC0        + $7BCF        + $DEA9
```

4. Perform the following hexadecimal subtractions and determine the validity of the result:

```
  $F562      $BACD      $DE35
- $C453    - $2FFE    - $F17D
```

5. Find the 2's complement expression in 8 bits of the following decimals:

```
 +49
 +63
+127
-128
 -59
 -37
```

6. Determine whether the following operations will give valid solutions for 8 bits **without** performing the calculations:

   (a) $1110\,1000_2 + 0010\,0001_2$

   (b) $1010\,1111_2 + 0100\,0001_2$

   (c) $0100\,1000_2 + 0100\,1000_2 + 0100\,1000_2$

7. Perform the following operations:

```
  1111 0000          0000 0001
+ 0111 1111        - 1111 1111


  0111 1111          1110 0011          1010 1001
- 1111 1111        + 0111 1110        - 0111 0001
```

   (a) Determine the V and C bits for each example and determine whether the operations are valid as signed and unsigned numbers for 8 bits.

   (b) Find the decimal expressions of all the numbers and the results above.

8. Convert the following numbers into *8-bit 2's complement* form:

   (a) 26_{10}

   (b) -73_{10}

   (c) -128_{10}

   (d) unsigned $0110\,1100_2$

9. Find the opposite of the following numbers in the *8-bit 2's complement* form:

   (a) 34_{10}

   (b) $0110\,1100_2$

   (c) $1001\,1001_2$

10. Perform the following arithmetic calculations in *2's complement*, and determine whether each solution is valid or not for 8 bits:

(a) $1101\,1111_2 + 1111\,1111_2$

(b) $0111\,1111_2 - 1111\,1101_2$

(c) $0000\,0000_2 - 1111\,1111_2$

(d) $1000\,0000_2 + 0111\,1111_2$

(e) $1010\,1010_2 - 0101\,0101_2$

11. Convert the following numbers into decimal assuming *unsigned* data, convert them again assuming *2's complement* data, and then determine for each form whether the solution would be valid for 8 bits **without** performing the calculation:

(a) $1100\,0111_2 + 1000\,1000_2$

(b) $1111\,0101_2 + 1110\,1011_2$

(c) $0111\,1101_2 + 0101\,1001_2$

(d) $1110\,0111_2 - 1111\,0001_2$

2.7 Laboratory exercises

1. Connect to WebLearn and complete Quiz 2. You should do this at least 5 times so that you get exposed to most of the questions in the quiz bank. (A different version of the quiz will be generated each time.) Discuss your answers with your lab assistant if needed.

2. Use any remaining time to seek assignment-related help or ask other course-related questions.

Chapter 3

Computer Organisation: The 68K Processor

Objectives

After studying this chapter, students should be able to:

- Describe the fetch, decode and execute cycle
- Discuss the main characteristics of RISC and CISC computers
- Discuss the different layers of abstraction of a computer system
- Describe the different types of memory technology
- Explain the different types of memory hierarchy, including cache and virtual memory

3.1 The Central Processing Unit

The CPU is where almost all of the operations of the computer are performed. The CPU includes a set of registers where information such as data and addresses may be stored while processing, and it has access to the buses to transfer information to and from memory. The main components of the CPU are (see Figure 3.1):

ALU, Arithmetic and Logic Unit: performs the arithmetic and logic functions, including `add`, `multiply`, `compare` and `branch`

Register Set: is for the internal storage of the current data being manipulated

CPU Buses: Transfer data from register to register, registers to memory, registers to ALU, and ALU to registers

MAR, Memory Access Register: stores the memory address to be accessed next via the address bus

MBR, Memory Buffer Register: also known as MDR, Memory Data Register. Stores the data just read from memory or the data ready to be written to memory

Program Counter: stores the address of the *next* instruction to be executed

Instruction Register: holds the next instruction to be executed

Stack Pointer: keeps the location of the top of the stack, so stack operations may be performed properly

Instruction Decoder: Converts a program instruction into the sequence of operations that executes the instruction

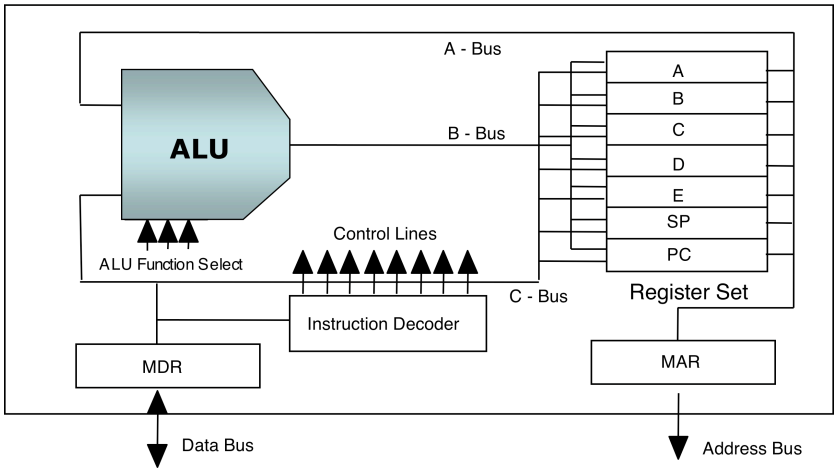

Figure 3.1: Main components of a CPU

To see how these operate, let's consider the instruction:
`'add E,B'`: Add the contents of register E to the contents of register B and save the result into register B.
Following the diagram in Figure 3.1, this instruction may be implemented as follows:

1. Transfer the contents of register E to an ALU input via the A-Bus.
2. Transfer the contents of register B to the other ALU input via the C-Bus
3. Select `add` as the ALU operation
4. Transfer the result from the ALU to register B via the B-Bus

3.1.1 Fetch, Decode and Execute

The standard computer cycle consists of continuously **Fetching** the next instruction and then **Decoding** and **Executing** the instruction. In each iteration,

1. The PC is moved to the MAR to access the memory location of the next instruction, and the CPU also sets the Read/Write line to read.
2. The MAR drives the address into the bus, and that is translated to a memory location that is then accessed.
3. The contents of the accessed memory location are copied onto the Data Bus and moved into the CPUs Instruction Decoder for decoding.
4. The Instruction Decoder decodes the instruction and uses its Control Lines to prepare to execute the instruction, by clocking data into registers, enabling registers onto buses, selecting ALU functions, and so on.
5. The ALU executes the instruction.
6. The PC is properly incremented to the address of the next instruction, ready for the next Fetch.
7. GOTO 1

3.2 CPU Instruction Sets: CISC vs. RISC computers

Complex Instruction Set Computers (CISC): In the beginning CPUs were very basic. Instructions and addressing modes were minimal, and much programming was done in assembly language. As chip technology improved, instruction sets and addressing modes expanded to make assembly language easier to write and supposedly more efficient. This meant that the instruction set of CPUs become bigger, as more and more instructions were created to satisfy the needs of programmers and compilers. Complex instruction sets produce shorter programs, and therefore use less memory, although there tends to be a proliferation of similar instructions, such as several different ways of accessing memory. As a consequence, instructions are of varying length, and some of them require many clock cycles — sometimes hundreds of clock cycles — to complete. In addition, CPU chips become extremely complex — including millions of gates — and this took considerable chip space, and the large number of gates generated a large quantity of heat that was hard to dissipate.

Studying programs running in computer systems, it was found that approximately 45% of the instructions are related to data movement, 25% to ALU operations and 30% to branching and flow of control. Over time, computer designers found that many of the complex instructions and addressing modes were infrequently used by compilers and users. In addition, the size of the complex logic reduced the number of registers available for executing programs, and the different instruction lengths meant a longer, more complex decoding cycle resulting in slower execution. There were many sub-routine calls, many memory accesses to save return addresses and parameters and returned results. Although some of the disadvantages may be overcome by high speed caches and wide and fast data buses (as we shall see later), the idea of producing simpler instruction sets took hold.

Reduced Instruction Set Computers (RISC): With the intention of simplifying the design and operation of the CPU, designers introduced processors with an instruction set in which all instructions are the same size, allowing the interspersing of

operations of different instructions — 'pipe-lining' — possible. Simpler instructions resulted in most instructions able to execute in shorter clock cycles — most execute in one clock cycle — so although there were more instructions they executed much faster. The Control Unit is much simplified, and as a consequence the silicon chip of a RISC is about a quarter the size of a CISC CPU, and generates much less heat. With less logic, designers can use some space for more registers — 256+ — so compilers and programmers can use them for temporary storage and parameter passing, therefore saving time by accessing registers rather than memory so frequently.

In addition, RISC processors are easier to design, hence requiring shorter design time, and therefore lower manufacturing cost. Speedwise, a slower RISC CPU can outperform a faster CISC processor. Some extra features, such as a Vector Processor may be added to improve processing speed for complex tasks such as graphics tasks; for instance, the PPC G4s can manipulate eight element arrays at once. Although the RISC approach makes the work of a compiler more complex this is not important to the programmer, and although the code produced is about twice the size of equivalent CISC code, it usually runs faster.

3.3 Computer Hierarchy: Layers of Abstraction

A computer offers users services by the execution of applications such as spreadsheets and wordprocessors. These are written in high-level languages such as C, C++ and Java, since these offer the programmer a development platform that is easier and safer (i.e. less error prone) to use. A compiler translates these programs into assembly or machine code, invoking the services of the Operating System — UNIX, Windows, etc — to perform its tasks. When running, the CPU instructions executed by the program are implemented within the processor as microcode procedures, which rely on the underlying digital logic to produce the necessary operations. (See Figure 3.2).

As it may be seen in the previous discussion, a computer operation is organised in a layered fashion, each lower layer offering a specific set of services to the layer above, in

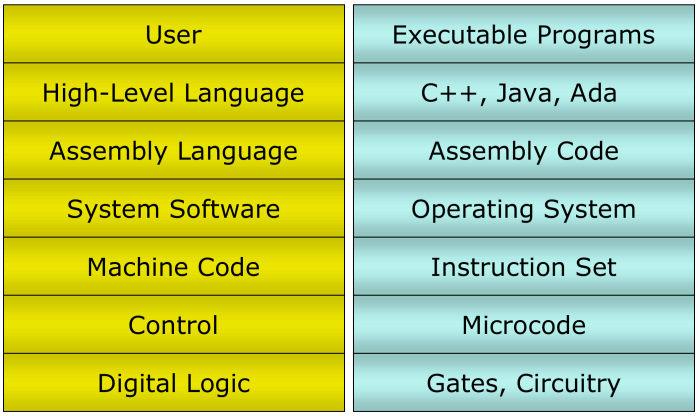

Figure 3.2: Computer Hierarchy

this way providing a safer executing environment. This relieves application programmers from the need to understand how to perform machine-dependent tasks, such as allocating memory, controlling memory access, managing disks and handling of files. An Operating System is made available for each type of computer to provide these services to higher layers, and the OS itself relies on services from lower layers to implement its services. In addition to being safer, writing applications in a higher-level language makes them portable, since the high-level code that runs on one machine may be (mostly) re-used after recompilation to run on a different machine.

Each layer is given a set of access points or functions, an Application Programming Interface (API), to be able to execute lower-level services. For example, an application does not access the computer I/O hardware directly; that would be too dangerous. Instead, a 'call' is made to software in the Operating System to supply the I/O service. This approach makes these operations safer, and hides lower layer details from the upper layers, hiding differences between computers making it possible for the same high-level application to run on different hardware. For example, a typical operation such as an array memory allocation in C or Java may work as follows:

1. an application programmer allocates an array within the code by declaring the array, say

   ```
   int myArray[10];
   ```

2. during compilation the compiler includes a request to the OS — using a system call such as `memget()`— for contiguous space within the program execution space

3. when the program runs, the OS attempts to satisfy the program memory request (it may fail!)

4. if successful, the CPU accesses the array memory locations as required by the program

5. when the program runs, at all times the OS protects each area of memory against accidental overwriting, and controls that each memory access is legal

3.4 Types of Memory Technology

There are essentially two different types of core (main) memory technology:

- Static: faster memory, using Flip-Flops (we shall see Flip-Flops later). It is faster to change state between 0 and 1 — to *switch state* — and it keeps state until the power is off, but it is more expensive.

- Dynamic: slower memory using capacitors. It is slower to switch, but it is possible to fit more units into a given space, it is cheaper to manufacture and it generates less heat. The problem is that capacitors lose their charge very quickly, so there is a need to refresh the contents after a few ms, otherwise they go to zero and lose the information.

Naturally, the faster the memory is, the faster the computer can perform its operations. However, other variables also contribute to the efficiency of the computer, such as:

- the speed of the CPU, since the faster the CPU the more operations it can perform in a given time interval

- the switching speed of memory

- the amount of memory available, because a larger portion of the running program will be available in memory ready to run (see Virtual Memory below)

- the width of the bus (to avoid multiple transfers for one memory access)
- the quality of the peripheral cards such as video and network cards, since they take load off the CPU

In general, the basis measure of computer performance is time. If we are only concerned with CPU performance (ignoring I/O issues for example), the processor dependent time is given by the formula:

$$ExecTime = NumOfInstructions \times AveClockCyclesPerInstruction \times ClockCycle$$

A computer with a smaller clock cycle, fewer clock cycles per instruction or with a smaller number of instructions to do a job will have a better performance.

There are are several measures of computer performance: two of the most common ones are:

MIPS: Million of instructions per second. This makes sense, but since instructions are so different from machine to machine, it does not give a very accurate measurement. For example, a RISC machine will execute more instructions per second than a CISC machine, but the CISC machine will need to execute fewer instructions to do the same job.

MFLOPS: Megaflops, millions of floating-point operations per second. This is an improvement over MIPS, since the measure gives an indication of how fast complex floating-point operations run on the processor, and therefore how efficient is the processor in getting operands, performing loops and other calculations and storing the results.

3.5 The Memory Hierarchy

Since some types of technology are more expensive but more efficient than others, to provide a better service memory in a computer is usually organised as a hierarchy, with faster, more efficient memory reserved for frequently used operations, and slower less efficient memory for less often used accesses. It is customary to classify memory according to how close they are to the CPU, with closer memory more efficient than more removed memory. Table 3.1 depicts the relative sizes for personal computers, costs and access times of the different types of memory in decreasing order of efficiency.

Table 3.1: Memory Hierarchy and Technology

Memory Type	Size (PC)	Access Times	Technology
Registers	1K, 2K, 4K bits	1-2 ns	SRAM
Cache Level 1	8K-16K	3-10 ns	SRAM
Cache Level 2	256K-512K	20-40 ns	SRAM
Main Memory	512K-1G	30-90 ns	DRAM
Hard Disk	80G-200G	5-20 ms	DRAM

3.5.1 Cache Memory

Since the slow access time to memory has a negative impact on the response time of the computer, and fast memory is very expensive, to speed up CPU instruction and data fetches from memory, and hence speed up CPU instruction throughput, processor designers often use very fast memory called *cache memory*. Cache memory is used to store a copy of the data and instructions as they are fetched by the CPU. Thus, when the CPU executes a program loop — such as a `for` or a `while` — or repeatedly accesses the same memory region — for example, when accessing the same variable set — the information is likely to be in the cache, which then can supply the information very quickly.

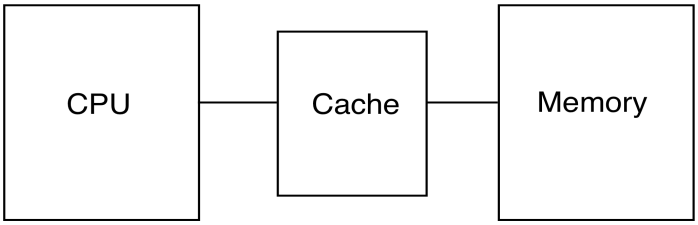

Figure 3.3: Cache memory

3.5.2 Virtual Memory

As we can see from the discussion, for the CPU to execute an instruction or to operate on a data item they need to be somewhere in memory. However, the size of modern programs means that usually a whole program does not fit in memory all at once. Hence, only the relevant parts of a process (a running program) are loaded into main memory, with the remainder residing on disk. When a program requests data that is not in main memory — e.g. an instruction, or data from a file — it has to get it from disk. This is organised transparently by the OS and the corresponding software and hardware, so programmers write programs as if the size of the computer memory is unlimited. Since this arrangement provides programmers with a virtual computer memory to work with, it is called *virtual memory*. The access times in Table 3.1 give an indication on the relative efficiency of the different accesses.

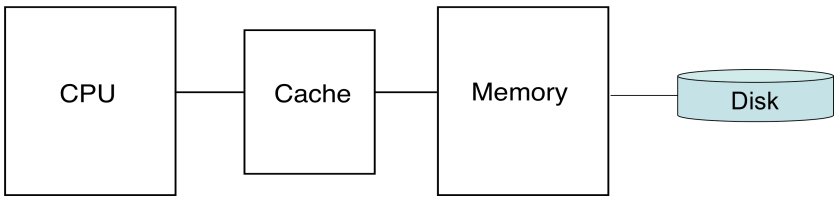

Figure 3.4: Virtual memory

To see how this works, assume a word is requested by the CPU; the word:

- is first searched for in the cache/s (primary level memory)
- if the word is in the cache (a *cache hit*), it is loaded immediately;

- if it is not in the cache (a *cache miss*), it is loaded from main memory (secondary level memory) if it is there;
- if it is not in main memory either, it must be brought in from disk.

With virtual memory, programs are written as if the memory capacity is only limited by the available capacity of the hard disk. However, when the required data is not in memory there is a significant performance penalty since I/O devices are several orders of magnitude slower than the processor and memory. That is why having a larger memory makes the computer run faster, since the OS may keep a larger portion of the process data in memory at one given time.

The most common way to implement virtual memory is by dividing up main memory into fixed-size blocks called *page frames* usually 1K, 2K or 4K in size, and dividing programs and data also into *pages* of the same size. When the required data is not in memory, the corresponding page where the data is in secondary memory is brought in, and placed in one of the frames, and accessed there.

When the data is in memory, the running program — the *process* — waits for the transfer via the data bus. When data is required from disk however, the CPU cannot block processing until the data comes in, since the CPU would be idle for too long. Instead, the running process is put to sleep — it cannot run anyhow since the data it needs is not available — waiting for the transfer of the data from disk, as follows:

- a whole page is requested to be transferred from disk;
- the incoming page is loaded into memory where it is now available for access.

Of course, this means that contiguous program pages not necessarily result in contiguous page frames in memory. Virtual memory works by translating each *logical* (*virtual*) memory address into a *physical* memory address. This is performed at run time, taking into account where in memory is each block located. In this way it is possible to:

- write programs as if there was only one user using the computer
- write programs independently of the memory capacity of the computer

All this operation is managed by the Operating System, which is in charge of determining whether the data is in memory and, if not, of locating where the data is on disk, and:

1. putting the process to sleep
2. if there is not enough room in memory, deciding which page is to be replaced to make room for the incoming page
3. issuing the read transfer
4. loading the page into memory (this accepts variations)
5. waiting until the transfer has finished

When the transfer is finished, the OS decides which process is going to run in the CPU next — it may not be the original process now sleeping — and dispatches this process to run in the CPU by copying all the process information into the CPU structures (registers, PC and IP).

Practical Work 3

3.6 Tutorial exercises

1. Draw a diagram of the main components of a computer, and describe each component
2. Investigate how many registers the 68K has, and state their characteristics
3. Describe the events/steps that occur during the Fetch/Execute cycle of the CPU. Include in your description: Program Counter, Data and Address buses, Memory, Control Unit, etc.
4. Find out the structure of a `move` instruction for the 68K, including all its variants, binary code, possible arguments, etc. Trace the Fetch-Decode-Execute cycle of a `move` instruction for the 68K.
5. The 68K is a CISC processor. Find an example of a RISC processor and draw a comparison between the two, in terms of length of instruction, types of instructions, number of registers and direct manipulation of elements in memory.

3.7 Laboratory exercises

1. Your assignment workings must be submitted in **PDF format** (see page 213). Therefore, you should use this time to practise generating PDF files. Refer to the Assignments section (page 216) for details on how to do this, and consult your lab assistant if you encounter any problems. (**Note:** both StarOffice and OpenOffice can directly generate PDFs from either a Word document or from their open standard ODF documents.)
2. Connect to WebLearn and complete Quiz 3. You should do this at least 5 times so that you get exposed to most of the questions in the quiz bank.
3. Use any remaining time to seek assignment-related help or ask other course-related questions.

Chapter 4

68K Structure and Assembly

Objectives

After studying this chapter, students should be able to:

- Describe the basic structure of the 68K chip
- Discuss the stack structure and its functionality
- Describe the structure of the 68K chip
- Explain the most commonly used instruction of the 68K instruction set
- Create, assemble and run a simple assembly language program using the 68K simulator
- Explain the concept of memory alignment
- Compare the two storing strategies big endian and little endian

4.1 Assembly Language Programming

Assembly language is closely related to the processor under consideration. Thus, when writing assembler language programs the programmer needs to know four fundamental things:

The problem to be solved: This is true for programming in general, not only for assembly

The Instruction Set of the CPU: That is, what instructions are available to the programmer

The CPU Register Set: What registers are available to the programmer to temporarily store and manipulate information

The CPU Addressing Modes: What are the different ways of accessing data in memory

For this course we are using the Motorola MC68000 Microprocessor family (there are several variants). This processor has many of the features present in commonly used computers and it is easy to use. Although each processor has particular characteristics, learning the 68K assembly language will make it possible to learn other assemblies if necessary.
The question is often asked, "Why are we learning assembly language (which few people program anymore) and why assembly for a processor that no one uses any more?"
There are several reasons for this, but the primary reasons are:

1. The 68K series of processors use a (more or less – mostly more) human readable representaion of numbers stored in the system. Several commonly used CPUs use representations ranging from difficult to "*just plain evil*". If you really need to program one of these CPUs you will learn to live with the way they store data – we do not need to for this subject.

2. The 68K series has a more rational instruction structure than some of the other common CPUs. Again this makes it easier for beginners to se what is happening.

3. While it is unlikely that many students will go on to full-time assembly language programming, an understanding of the relationship between common "high-level" language code constructs and the assembly (machine language) code required to implement the construct can improve the efficiency of programs. (Admittedly, efficiency is not a major goal for most contemporary programmers, but a good argument could be made that it *should* be a goal.)

4. A fair bit of the "embeded" program code (e.g. the program code in your dvd player

or microwave oven) is still being written in assembly language. (Primarily to keep the size to an absolute minimum)

Instruction Set: The 68000 has about 60 instructions, each with several variations. We are going to use only a reduced set that will be enough for our purposes.

Instruction Format: All instructions follow the format:

```
operation.size    source,destination
```

where some of the operands may be implicit. For example:

```
add.w    d0,d1
move.l   d1,a0
bne      next
```

are all legal 68K instructions.

Allowed operations include `add`, `sub`, `mul`, `div`, `move` ... The Source and the Destination are the locations of the operands, that may be an address in memory or a CPU register. The data size is either:

- b = byte, 8 bits
- w = word, 16 bits
- l = long word, 32 bits

NOTE: Individual bits in memory are not addressable, the smallest addressable unit is a byte.

The Register Set: The 68000 has seventeen 32-bit registers and a 16-bit Status Register. Figure 4.1 illustrates the register set of the 68K processor.

- Data Registers: There are eight general-purpose data registers, named `d0` to `d7`

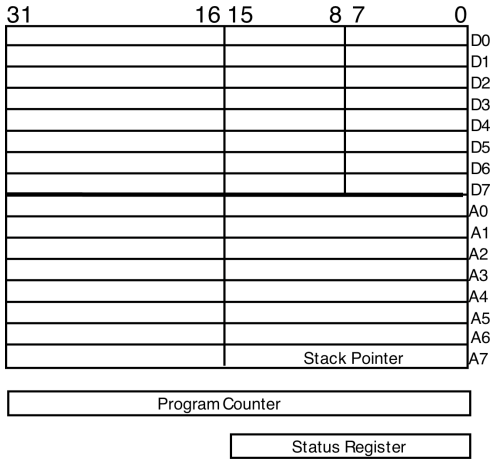

Figure 4.1: Register set of the 68K processor

- Address Registers: There are eight address registers, named `a0` to `a7`, specifically to store addresses. Register `a7` a is special register: it is called the Stack Pointer (often referred within assembly as `sp`), and it used to indicate the position of either the user or system stack (more on this later).

Program Counter: stores the address of the next instruction, the instruction to be executed next.

Status Register: comprised of a user byte — the Condition Code Register (CCR) — and a system byte.

4.1.1 The Stack

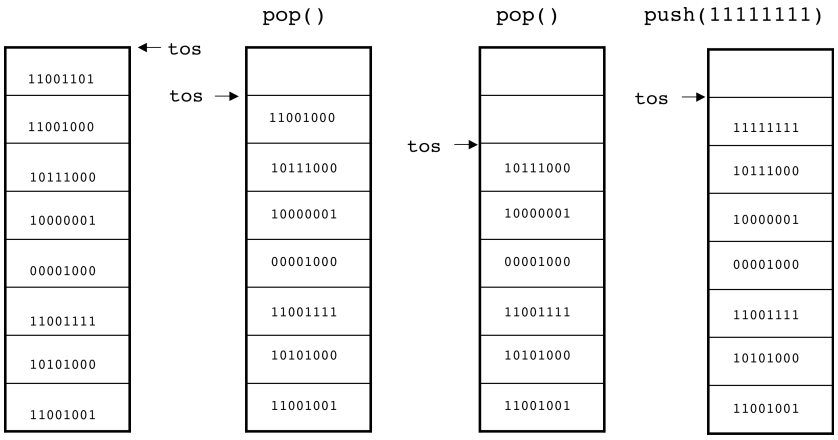

Figure 4.2: The stack structure

A stack is a set of memory locations used to store information temporarily. Stack locations are sequential locations (side by side, like an array), and they are managed at one end, the *top of the stack*, which is pointed to by the stack pointer. A stack is a LIFO (Last In First Out) structure, since the last element going onto the stack is the first element to come off the stack. The only operations allowed at the top of the stack are:

- `push()` an element on the stack: put it on top of the stack
- `pop()` an element from the stack: take it off the top of the stack

A stack is called so because it is similar to a stack of plates in a cafeteria, where the last plate to be put on top (pushed) is the first one to come off (popped). (See Figure 4.2.)

4.2 The 68K Instruction Set

The following tables summarise the most common instruction of the 68K instruction set.

Table 4.1: Data Movement Operations

Instruction	Name	Data Size	Operation
`exg`	exchange	32	Rx ↔ Ry
`move`	move b, w, l	8,16,32	(EA)s → EAd
`swap`	swap words	32	W ↔ W

Table 4.2: Compare Operations

Instruction	Name	Data Size	Operation
`cmp`	compare	8, 16, 32	Dn – S, condition bits set
`cmpa`	compare address	16, 32	An – S, condition bits set
`cmpi`	compare immediate	8, 16, 32	D – #data, condition bits set
`cmpm`	compare memory	8, 16, 32	D – S, condition bits set

Table 4.3: Integer Arithmetic Operations

Instruction	Name	Data Size	Operation
`add`	add	8, 16, 32	S + D → D
`clr`	clear	8, 16, 32	0 → EA
`cmp`	compare	8, 16, 32	D – S
`divs`	divide signed	32 div 16	D div S → RQ = D
`divu`	divide unsigned	32 div 16	D div S → RQ = D
`muls`	multiply signed	16 X 16 → 32	S x D → D
`mulu`	multiply unsigned	16 X 16 → 32	S x D → D
`neg`	negate	8, 16, 32	2's complement
`sub`	subtract	8, 16, 32	D – S → D

Table 4.4: Branching Operations

Instruction	Name	Operation
`bra`	branch always	PC + d → PC
`bsr`	branch to subroutine	PC → (SP), PC+d → PC
`bcc`	branch on condition	if TRUE then PC+d → PC
`beq`	branch if equal	if E then PC+d → PC
`bne`	branch if not equal	if NE then PC+d → PC
`bgt`	branch if greater than	if GT then PC+d → PC
`bge`	branch if greater than or equal	if GE then PC+d → PC

Table 4.5: Bit Manipulation Operations

Instruction	Name	Data Size	Operation
`btst`	bit test	8, 32	bit tested → Z
`bset`	test bit and set	8, 32	1 → bit
`bclr`	test a bit and clear	8, 32	0 → EA
`bchg`	test a bit and change	8, 32	toggle bit

Table 4.6: Logical Operations

Instruction	Name	Data Size	Operation
and	logical and	8, 32	S and D → D
or	logical or	8, 32	S or D → D
eor	exclusive or	8, 32	S eor D → D
not	logical not	8, 32	not(EA) → EA

4.2.1 Sample Program

The following example is an assembly language program to add together the data bytes at memory locations `$3000`, `$3001`, and `$3002` and put the result into location `$3003`. This is the equivalent of the code `result = data1 + data2 + data3`. (See Figure 4.3.)

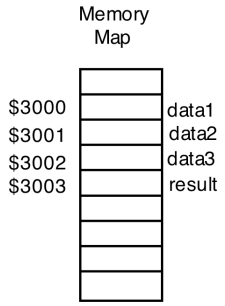

Figure 4.3: Memory map for sample program

Code Example 4.1: Add values from 3 memory locations

```
1       org     $1000
2       move.l  #$7ffe,sp       ;initialise stack pointer
        move.b  $3000,d0        ;get data from $3000
4       add.b   $3001,d0        ;add the next data
        add.b   $3002,d0        ;add next data
6       move.b  d0,$3003        ;save result
        move.b  #9,d0           ;return to simulator
8       trap    #15
        end                     ;end of code
```

- The `org` directive indicates to the assembler that the code is to reside in memory, starting at location `$1000`.
- Next you will find the instruction mnemonics — `add`, `move` — each followed by the source and destination addresses. Except for the first `move` which is long word `.l`,

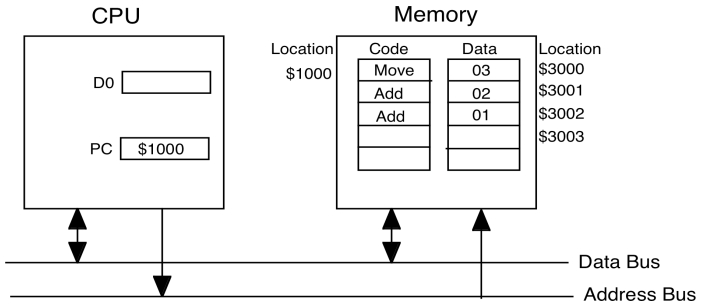

Figure 4.4: Running the sample program

the data size for each instruction is byte, so the instructions end in `.b`. Each part of the instruction line is delimited by spaces or tabs.

- The final part of the instruction is a comment, which describe the action taken as part of the problem solution (not a description of the instruction itself). The general instruction format is:

```
mnemonic.size   source,destination      ;comment
```

Figure 4.4 depicts the computer status as it runs the sample program.

NOTE: Later on we will see how to create and run this program, but when creating this code *you must put one or more spaces at the beginning of each line.* These may be replaced by tabs.

Let's trace the execution of the instruction:

```
move.b   $3000,d0       ;get data from $3000
```

As always, the instruction cycle is:

FETCH: The address in the PC goes to memory via the address bus. The instruction in that location is transferred to the CPU instruction decoder via the data bus.

DECODE: The `move` instruction is decoded. The instruction and its parameters are identified.

EXECUTE: The steps are:

1. The CPU sends address `$3000` to memory via the address bus
2. Memory transfers the data in memory location `$3000` to `d0`, via the data bus
3. The PC is incremented and the new cycle begins for the next instruction

The program continues with the execution of the two `add` instructions. The contents of data register `d0` will change from 03 to 05 then to a 06 as the program progresses. The next instruction:

```
move.b     d0,$3003
```

puts the address, `$3003` onto the address bus and the data in `d0` onto the data bus. The data will then be stored in location `$3003`.

The two instructions:

```
move.b     #9,d0
trap       #15
```

exit the execution and return control to the user (more on this later).

4.2.2 Assembler Directives

Assemblers have their own set of instructions called *directives* to control the assembly process:

`org` : indicates where the code is to start in memory. The first part of memory from `$0000` to `$0FFF` is reserved for the use of the CPU to handle interrupts (more on this later). Therefore, user programs can start at `$1000`. The end of user memory is at `$7FFF`. This gives the user about 31K of RAM to use.

`dc` : 'define constant' is a means of putting data into memory at the time that the code is loaded:

```
dc.b    8
dc.w    $1234               ;define integers
dc.b    'computers'         ;defines a string
```

The data is stored in memory at the current location counter (i.e. the location of the instruction). String characters are stored in consecutive memory locations.

`ds` : reserves space in memory for data, but does not put any data into the location. The reserved memory is at the current location counter.

`end` : indicates that there is no more code to assemble.

An assembler directive looks like an instruction, but it is not an instruction. Directives tell the assembler how to assemble the code, so their effect takes place at assembly time, *before* the instructions are decoded and executed. That is, an assembler directive does not go through the normal Fetch, Decode, Execute cycle as an instruction does.

4.3 The EASy68K Motorola 68000 Simulator

To write, assemble and run a 68K programs on a PC the EASy68K simulator is used. This is a program designed to simulate the behaviour of a Motorola 68000 series CPU with a basic operating system. It is available from `http://www.easy68k.com/` and is installed on the Blowflies as well as the Windows machines in the School's Labs.
For those using either Linux or MacOS, there is a product called CrossOver available that gives you access to the full features of EASy68K, including the Windows Help file. (A Student/Education discount has been negotiated. Note that the more expensive version is also supposed to run a number of Windows games in Linux – I have not tested this claim, though.)
Notes on how to install EASy68K under CrossOver will be available from Blackboard at a later date.

To create and run your first program, start the EASy68K editor/Assembler. You will see a window that looks something like 4.5. Enter the program code in the window as required. The bits at the top are reminders of information that is useful to include with every program. To make this more useful, look for a file called `template.NEW` in the EASy68K directory. It would be a good idea to copy the line that says: `* Written By:` and replace the text with `* Student No:` to remind you to always insert your student number in work you are submitting. **Note:** a Line that begins with a '*' (star or asterisk) is a comment line. The assembler *will* complain if you leave out the star.

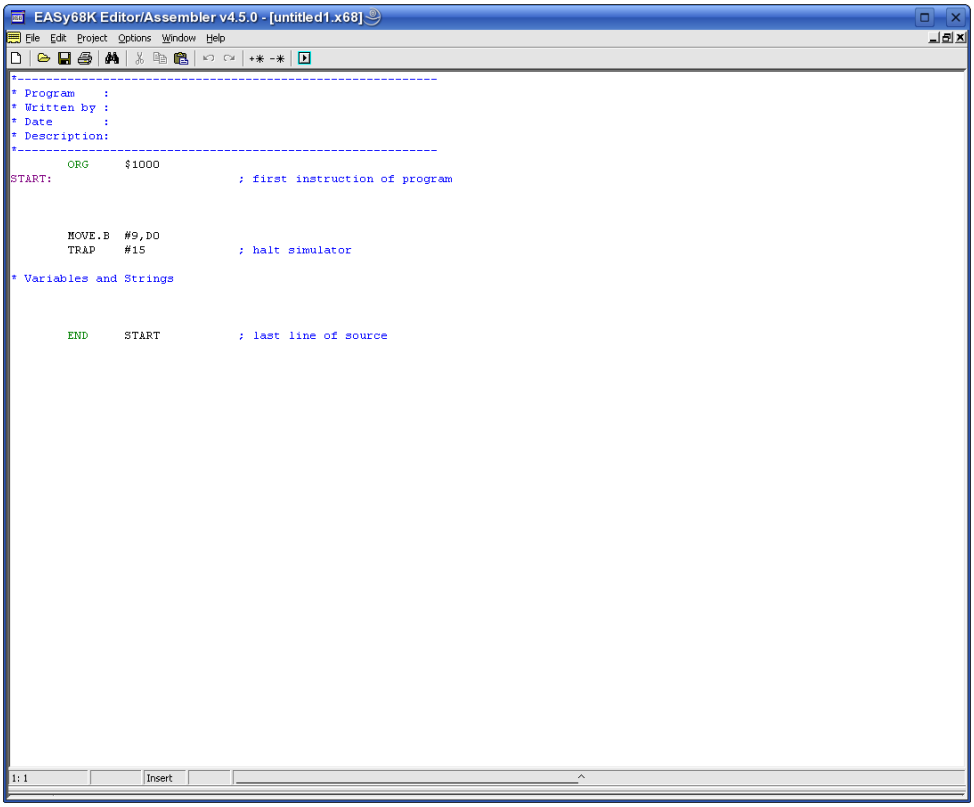

Figure 4.5: The EASy68K Edit Window

In the labs you will first use the Edit68K Editor/Assembler to enter your program and assemble it. After you have entered your code, you can either click on the button that looks like a DVD "Play" button, or in the Project menu, use the Assemble option, or use the F9 key as a shortcut.

The simulator is launched by clicking on the "Execute" button in the window that is displayed after a successful assembly. (See figure 4.6.)

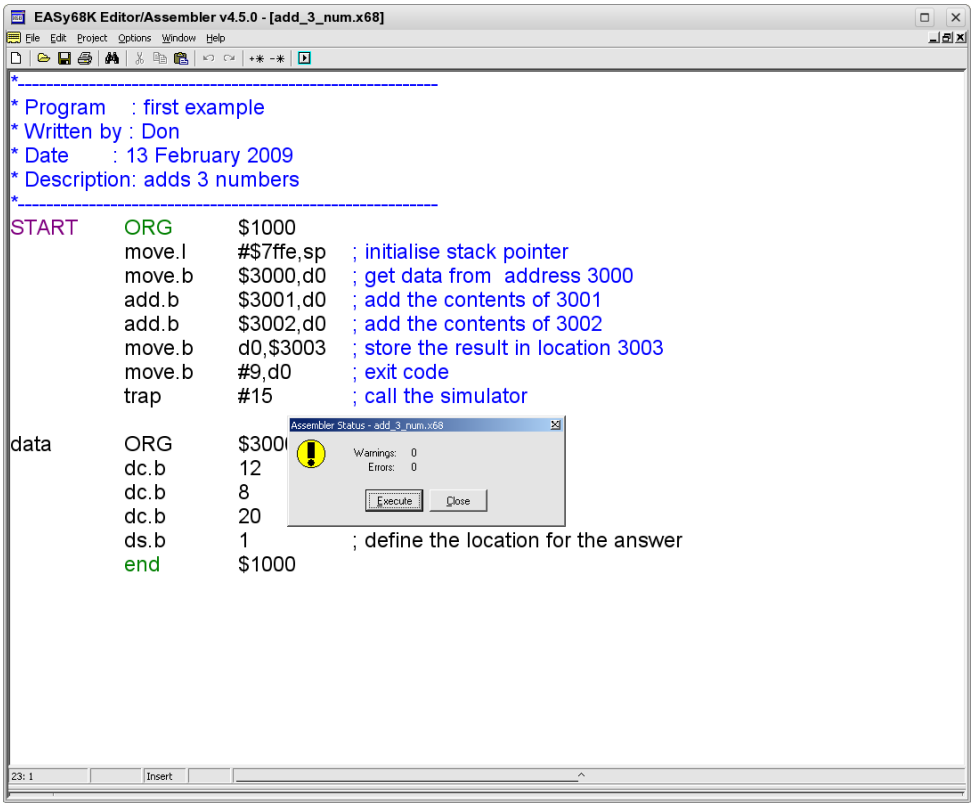

Figure 4.6: After assembling the program

The execution windows will open. The initial window, shown in figure 4.7, shows the source code and the contents of the registers. The buttons allow you to run your program in various ways. The simplest is pressing the "Play" button that runs the program until it either completes or has an error condition.

Looking at the buttons in the simulator and what they do.

The leftmost button allows you to load a different program. If the program that you want to run was previously written and assembled, there will be a file with an '.S68' extension. This is "machine code" for the 68000. It can be interpreted by the simulator.

The next button is the "Play" button to run the program to completion.

The next button, which says "Run to cursor" if you put your mouse on it, allows you

to highlight a line in the source code and run the program with a stop at that line. This can be particularly useful with a loop, where you want to check the status of a number of aspects of the program at run-time. By highlighting a line near the end of the loop, you can check the changes that occur with each iteration.

The "Auto Trace", "Step Over" and "Trace Into" options will become clearer as we start to use more advanced programming techniques. In particular, "Step Over" does not follow the flow of execution into subroutines, while "Trace Into" does follow the flow. (This will be discussed again later, so do not be too concerned about it now.)

The execution environment gives you the option of seeing the "terminal" where the text I/O is occurring, as well as the memory (in hexadecimal) and the contents of the stack. To select what is shown, use the "View" pull-down menu. Figure 4.8 shows our begining program running with an I/O window and a memory inspection window.

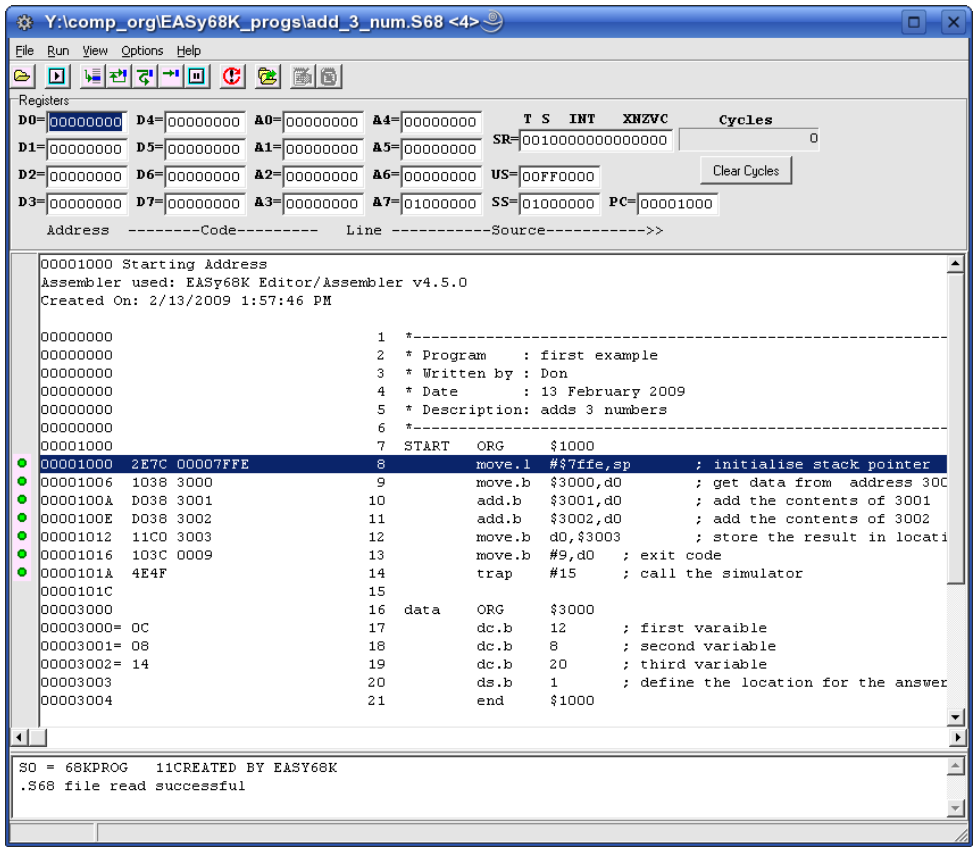

Figure 4.7: The Main Run-Time Window

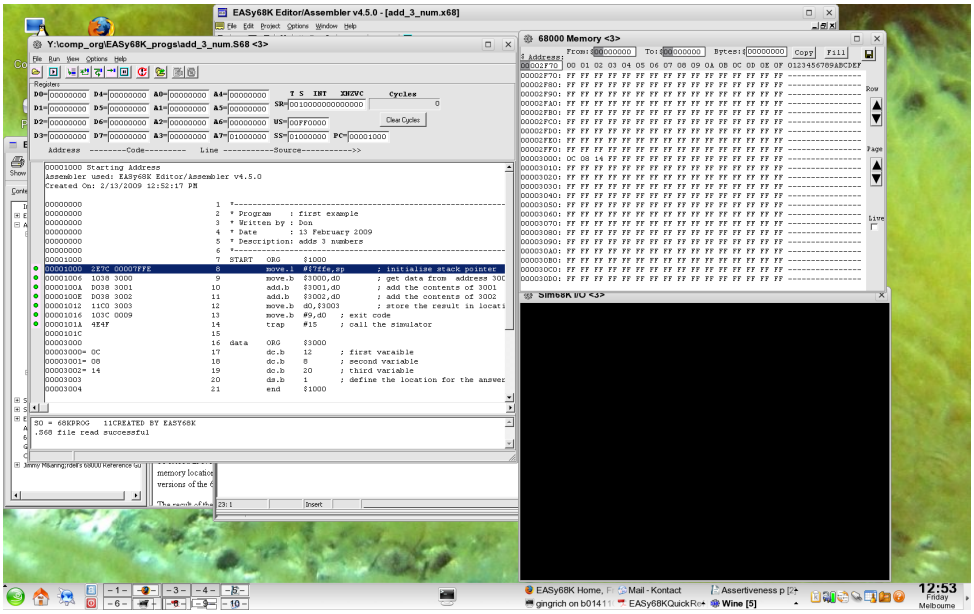

Figure 4.8: Full Run Time Environment

4.3.1 EASy68K - Run time errors

The most common run time error messages are *bus errors*, when trying to access an illegal address (for example, above `$7fff`, which is the top of memory) and also address errors when the CPU attempted to access a word or long word at an odd address (more on this later). (see figure 4.9) You will note that the EASy68K assembler has automatically fixed the problem of the word that follows the byte at address 3000. The word is stored at an address of 3002. (see line 18 in the display.) But our code in the incorrectly written program is still trying to `move` the word from address 3001, so there is an error. A more advanced compiler for a high level language would manage this automatically. To de-bug a program, study the contents of the registers, immediately under the "control buttons". It can also be useful to look at the contents of memory. Note that you can move through the memory with the page and row up and down buttons. The instruction which caused the error will be the instruction that is highlighted in the run time window. The instructions (in hexadecimal), as well as the data and the addresses as set by the assembler are all visible at the right side of the run time window.

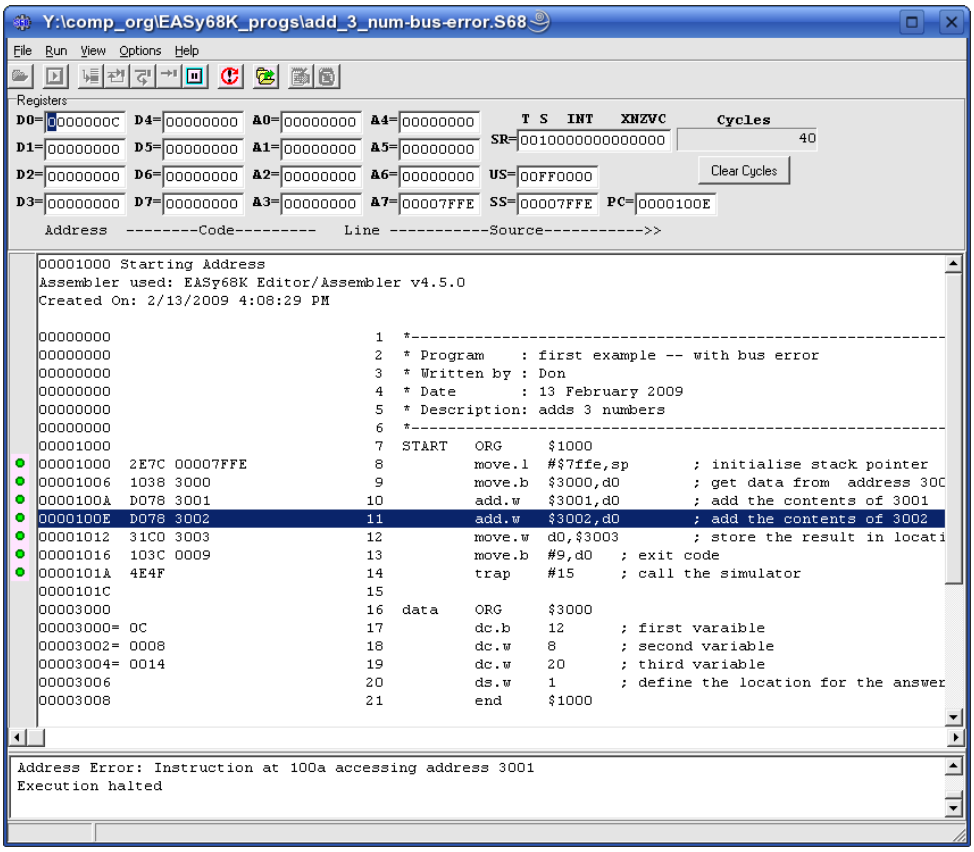

Figure 4.9: Program with an Addressing Error

4.4 Memory alignment

Although it is possible to address individual bytes in memory, often the processor moves around multibyte data, that is, (in the 68K) data made up of 2 or 4 bytes. Bytes may be placed anywhere in memory, but in the 68K both words and long words *must* start at even addresses. An attempt to place a word or long word at an odd location will result in a "Bus Error" message. Figure 4.10 illustrates the relative locations of several bytes, words and long words, properly aligned in memory.

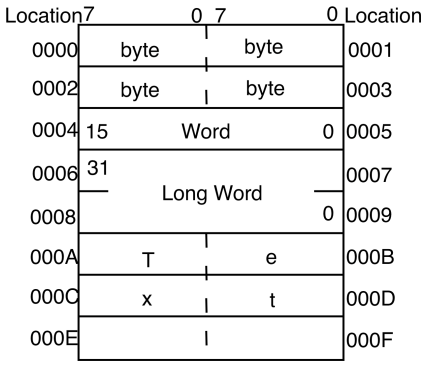

Figure 4.10: Multi byte data in memory

We can see in the figure that bytes can be at any address, but words and long words must be at even numbered addresses. Words use two consecutive locations, whereas long words use four consecutive locations. You can see in the figure that there may be some wasted space, but the availability of cheap memory has made this waste less important.

4.5 Endian

Multi-byte data can be stored in memory with the lowest location holding the Most Significant part as in the Motorola 68K, or with the Least Significant part at the lower memory location as in the Intel XX86 series microprocessors. Thus, a two-byte integer may be stored with the most significant byte first followed by the least significant byte — this is called *big endian* — or the other way around — *little endian* — where the

LSB is stored at the lower address. Hence, big endian machines store the MSB at the lower address, while little endian machines store the LSB at the lower address. Figure 4.11 illustrates the situation of the two approaches when storing a multi-byte quantity.

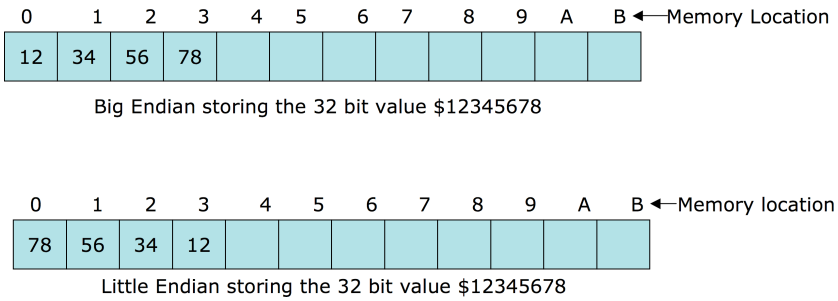

Figure 4.11: Endian

There are advantages and disadvantages with both methods, but they are not too significant. Big endian is easier to read for humans, and this is very important for the interpretation of the 68K simulator dump. The following example for a 4-byte integer illustrates:

byte 3	byte 2	byte 1	byte 0

On a little endian machine it would be stored as:

base address + 0 = byte 0
base address + 1 = byte 1
base address + 2 = byte 2
base address + 3 = byte 3

On a big endian machine it would be stored instead as:

base address + 0 = byte 3
base address + 1 = byte 2
base address + 2 = byte 1
base address + 3 = byte 0

4.6 Sample Program

Create and assemble the following program in the Editor/Assembler.

Code Example 4.2: Add values from 3 memory locations

```
1       org     $1000
2       move.l  #$7ffe,sp       ;init stack pointer
        move.w  $2000,d3        ;get data 1
4       add.w   $2002,d3        ;add data 2
        add.w   $2004,d3        ;add data 3
6       move.w  d3,$2006        ;save result
        move.b  #9,d0           ;return to simulator
8       trap    #15             ;
        end                     ;no more to assemble
```

NOTE: Remember to put a space at the beginning of each line of the source code. Assembly language has a rather strict line syntax since it is one of the earliest computer languages and was designed to be easy to implement on machines with *very* little memory. The simpler the format, the easier it is to write a program to process it. And the smaller that program will be.

After you have successfully assembled the program, (No errors in the status box) use the "Execute" button to start the runtime environment. In the runtime environment, click on "View" to open the menu and select "Memory". When the memory window opens use the down arrow page button to move through the memory until `00002000` is visible in the numbers at the left side. (The memory window should look something like figure 4.12.) You will need to insert the numbers that you want to add. There are two important points to remember:

1. Each number occupies two bytes

2. The numbers are in base 16 (hexadecimal) and they comprise 4 hexadecimal digits. (Not quite the right word since digit is actually a base 10 or decimal concept) And hexidecits does not sound right either. Let us call them hexidecimal characters meaning a number between 0 and F.

By putting the cursor over one of the FF characters that mark uninitialised byte locations, you can type in the values that you want to add. In the example, the values are:

Address	Hexidecimal value	Decimal value
2000	0010	16
2002	0018	24
2004	0028	40

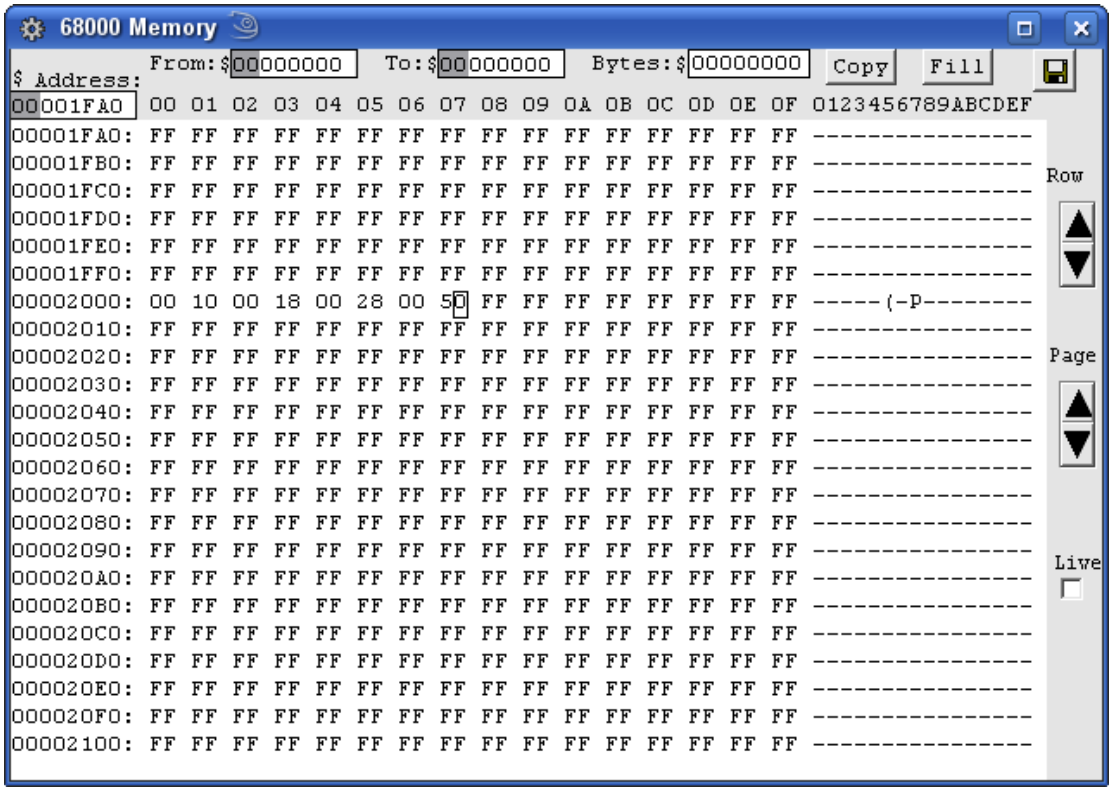

Figure 4.12: The memory map at address 0x2000

This sample program adds together the word size data at locations `$2000`, `$2002`, `$2004` and saves the result in memory location `$2006`. Note that each number is 2 bytes, because the data is word size.

Practical Work 4

4.7 Tutorial exercises

1. Describe cache and virtual memory, and explain how they are used by the system in manageing data.

2. What is paging? In one or two sentences describe the paging mechanism.

3. Explain how an Operating System manages a page fault.

4. By now you will have some experience programming in Java or C, and will have worked with various data types such as strings (an `Object` in Java; an array of `char`s in C), integers, floating point numbers, etc. As we know, all computer data is stored in binary, each data type requires a different amount of space in memory, and the binary data gets interpreted in different ways. The table below lists all Java primitive types, their memory requirements and interpretation.

Type	Space	Interpretation	Range
`boolean`	1 bit	`0` ⇒ `false`; `1` ⇒ `true`	`true/false`
`byte`	1 byte	2's complement	`-128` to `127`
`char`	2 bytes	Unsigned (Unicode)	Each code represents a separate character
`double`	8 bytes	IEEE 754	$(\pm)$ 4.94×10^{-324} to 1.79×10^{308}
`float`	4 bytes	IEEE 754	$(\pm)$ 1.40×10^{-45} to 3.40×10^{38}
`int`	4 bytes	2's complement	`-2,147,483,648` to `2,147,483,647`
`long`	8 bytes	2's complement	`-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`
`short`	2 bytes	2's complement	`-32,768` to `32,767`

In modern times where memory is cheap, most Java programmers rarely use the smaller primitives such as `short` and `byte`; they can however be used to reduce the program's space in memory if this is a concern.

We will see *Unicode* and the *IEEE 754* format for floating point numbers later in the semester, but the remainder of this tutorial will focus on the primitive types `boolean`, `int`, `short` and `long`.

For the purposes of this tutorial you should assume that *big endian* is being used (i.e. the most significant bytes come first). Boolean values are determined from the

least significant bit within a byte. Any data longer than a byte must begin at an *even* address.

A Java program is running in virtual memory. The following diagram shows the data held in the virtual memory in hexadecimal format. Note that each memory location holds one byte. The values in grey (the values from memory location `$2010` onwards) represent residual data (this memory is not yet allocated by the program).

(a) What is the *decimal* value of the following:

i. `byte` starting at memory address `$2003`

ii. `short` starting at memory address `$2002`

iii. `int` starting at memory address `$2000`

(b) What is the Boolean value stored at the following memory addresses:

i. `$2004`

ii. `$200B`

(c) The Java program allocates memory space, in order, to variables of the following types. Assume that the next available piece of free memory is allocated. On the memory diagram above, shade the blocks to indicate the newly allocated memory (use different shades to represent each data type):

i. a `boolean` called `success`

ii. an `int` called `sum`

iii. a `short` called `count`

(d) Change the data stored in memory to reflect the following operations:

i. `success = true;`

ii. `count = 128;`

iii. `sum = count + 27;`

5. Answer the following questions:

(a) How many 16-bit registers does the 68K CPU have? How many 32-bit registers?

(b) What is the `sp` register? What does it do? In what sense it is different to the other address registers?

(c) What is the `pc` register? What is its function?

(d) What are the data sizes supported in the 68K?

4.8 Laboratory exercises

1. Your lab assistant will provide you with a sheet of instructions for getting acquainted with the assembler and simulator. This sheet is also available for download from the course's Blackboard (Course Downloads → EASy68K Primer). Run through the instructions on this sheet and raise any questions with your lab assistant.

   Note: EASy68K is written as a Windows program, but it has been tested and will work in WINE or CrossOver under Linux. And it has also been tested in CrossOver in MacOS. WINE is free for installation in Linux. CrossOver is a commercial package based on WINE. It has some features that should make it easier to install and run Windows programs compared to WINE.

   If you are using Linux or MacOS, it is up to you how you manage this. Your tutor *may* be able to assist if you have problems, but running tools on multiple operating systems is ***not*** one of the requirements for tutoring this subject.

2. Once you are comfortable with EASy68K, code some of the sample programs seen during the lectures and tutorials, and assemble and run them. Learn what causes the assembler to reject your code, and remember this for when you are completing your assignments. Experiment with the simulator by tracing through the programs. Pay particular attention to the register contents and memory contents as the programs run.

Chapter 5

Data Representation (cont.)

Objectives

After studying this module, students should be able to:

- Explain binary multiplication operations, for unsigned and signed numbers
- Describe and correctly manipulate binary numbers with a fractional component
- Discuss the IEEE Standard 754 Floating Point Format, and properly find representations conforming with the format
- Describe alphanumeric codes such as ASCII and BCD, and convert between them and binary

5.1 Binary Multiplication

5.1.1 Unsigned Binary Multiplication

Binary unsigned numbers may be multiplied in a similar way to decimals. For example:

010 0011 * 101 = (0010 0011 * 100) + (0010 0011 * 1) =
= 1000 1100 + 0010 0011 = 1010 1111

We can see in the example that multiplying by 10 (i.e. multiplying by 2) is shifting 1 bit to the left and inserting 1 zero to the right. Similarly, multiplying by 100 is shifting 2 bits to the left and inserting 2 zeroes to the right, multiplying by 1000 is shifting 3

bits to the left, etc. So it is easy to multiply by shifting and adding after each shift.

As a consequence of the shifting, we can see that the multiplication may go beyond the range of acceptable numbers, as in:

1110 0011 * 101 = (1110 0011 * 100) + 1110 0011 = 100 0110 1111

The result of the operation above does not fit into an 8-bit number. In general, the multiplication of two 8-bit numbers may produce a 16-bit result, and the multiplication of two 16-bit numbers may produce a 32-bit result.

To multiply two binary numbers, we may use a similar algorithm to the one we use in decimal multiplication:

```
    1110 0011
 *        101
--------------
     11100011
    000000000
   1110001100
--------------
  10001101111
```

5.1.2 Sign Extension

As we can see in the previous section, often operations performed on 8 bits produce a result comprising 16 bits. Sometimes we need to extend a number from a lower to a higher number of bits — say 8 bits to 16 bits — to be able to perform an operation. This is called *sign extension*. Sign extension is a way of extending the size in bits of a number, without changing its value. Say we want 0110 extended to 8 bits; this is quite simple `0000 0110` nothing major happens since we have 0s to the left of the given

number, and the number does not change.

However, if we have 1101, we can also extend it to `1111 1101` and the value as a signed number does not change, as we can see in the following calculation

$$1101 = -2^3 + 2^2 + 2^0 = -8 + 4 + 1 = -3$$

and again

$$11111101 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0 = -3$$

This is true in general: if we extend with either 0s or 1s, the original number does not change.

5.2 Positive Binary Fractions

We represent positive decimal fractions by using negative powers of the base 10:

Position Value:	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
decimal	.1	.01	.001	.0001	.00001

Positive binary fractions may be represented in the same way as positive decimal fractions by using negative powers of 2:

Position Value:	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
decimal	.5	.025	.125	.0625	.03125
fractions	1/2	1/4	1/8	1/16	1/32

5.2.1 Conversion of positive binary fractions

To convert a decimal fraction to a binary one, we use the following algorithm. We repeatedly multiply by 2, copy and subtract the integer part until enough digits are obtained or a product is zero. For example, convert $\texttt{.637}_{10}$:

```
.637 * 2 = 1.274 [copy 1 down, and subtract 1]   1
.274 * 2 = 0.548 [copy 0 down, and subtract 0]   0
.548 * 2 = 1.096 [copy 1 down, and subtract 1]   1
.096 * 2 = 0.192 [copy 0 down, and subtract 0]   0
.192 * 2 = 0.384 [copy 0 down, and subtract 0]   0
.384 * 2 = 0.768 [copy 0 down, and subtract 0]   0
.768 * 2 = 1.536 [copy 1 down, and subtract 1]   1
.536 * 2 = 1.072 [copy 1 down, and subtract 1]   1
```

Answer: $\texttt{.637}_{10} = \texttt{0.10100011}_2$

An alternative method may be used, if we note that multiplying by 256 (= 1 0000 0000 binary) results in shifting the fraction point of a binary number 8 positions to the right, and dividing by 256 results in the fraction point shifted 8 positions to the left. Considering 8 decimal places, using the same example again $\texttt{.637}_{10}$, we multiply and divide by 256:

```
(.637 * 256)/256 = 163.072/256
```

and this is approximately

```
163/256 = 1010 0011/256 = .1010 0011
```

which is the previous result.

The same method may be used to convert from binary fractions to decimal fractions. Consider for example the binary number 0.0100 1001; this multiplied by $256 = 2^8$, results in 0100 1001, so:

```
0.0100 1001 = (0.0100 1001 * 256)/256 = 0100 1001/256 = 73/256 =
= 0.28515625
```

When a number has an integer part and a fractional part, both should be treated separately. For example, to convert $\texttt{59.637}_{10}$ to binary, we first split the number as $\texttt{59}_{10} + \texttt{.637}_{10}$, and then:

convert 59_{10} to binary: 00111011_2

convert $.637_{10}$ to binary (previously done): $.10100011_2$

Therefore: $59.637_{10} = 00111011.10100011_2$

Addition operations are also performed separately, taking care of the carry out of the decimal part into the integer part, as in:

$$00111010.10100011_2 + 01000100.10001100_2 = 01111111.00101111_2$$

Student Exercises

- Convert decimals to binary (8-bit max):

  `0.5`

  `0.75`

  `0.123`

  `17.994`

  `79.001`

- Convert first the numbers to binary and then perform the following binary operations (8-bits max):

  `17.75 + 79.001`

  `79.123 + 17.994`

Note: A critical point that you should take from these exercises is that, in many cases, it is impossible to exactly convert a decimal fraction to a binary fraction. This influences a number of aspects of a computer's handling of base 10 "real" numbers as opposed to base 10 integers. Integers convert precisely, the part of the number to the right of the decimal point generally becomes an approximation in binary.

5.3 Floats

Binary integers in a computer are of fixed and limited size, and therefore it is not possible to represent very small negative or very large positive values. Floating point numbers — numbers written in scientific notation — allow a much wider range of numbers to be represented. However, floating point numbers are only *approximations* of the actual value, so there is loss of accuracy with the approximate representation.

Format: a * r^{exp}, where a = mantissa, r = radix/base, e = exponent

In a computer, two fields are stored as depicted in Figure 5.1. The exponent and mantissa can use any of the binary systems. The radix is not stored as it is a known value.

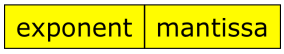

Figure 5.1: Floating point representation

Simple example

Consider the simple scheme illustrated by Figure 5.2

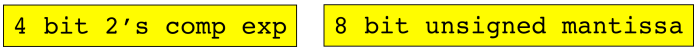

Figure 5.2: A simple floating point

and the simple example illustrated by Figure 5.3:

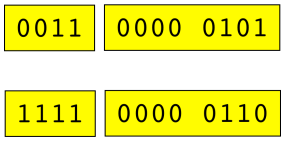

Figure 5.3: A simple example of floating point representation

Floats can represent a wide range of numbers, but there is almost always some error because of the distance between each number in the representation scheme. The

Exponent	Mantissa
0011	0000 0101 = 5 * 2^3 = 5 * 8 = 40_{10}
1111	0000 0110 = 6 * 2^{-1} = 6 * .5 = 3_{10}

conversion between the actual number and the computer representation is not always exact, even when storing an integer. That is, when an integer number is stored as a float, it may suffer from loss of accuracy due to the representation. The integer number 250 when stored as an integer is 1111 1010, an exact representation of 250. However, if try to to store 250 as a float we may get only an approximation, as we shall see in the next section.

5.3.1 IEEE Standard 754 Floating Point Format

To replace the individual non-standard representations used by the different manufacturers, in 1985 the IEEE introduced the IEEE Standard 754 Floating Point Format to represent floating point numbers, This IEEE standard is widely used in modern processors. There are three different precisions in use:

- Single precision: 32 bits in length, as follows:
  - 1 bit sign
  - 8 bit exponent in excess 127 format
  - 23 bit mantissa
- Double precision: 64 bits in length, as follows:
  - 1 bit sign
  - 11 bit exponent in excess 1023 format
  - 52 bit mantissa
- Extended precision: 80 bits in length, only used internally by the computer arithmetic unit to reduce rounding errors.

5.3.2 From Decimal to IEEE 754: single Precision

Let's see an example of how to store an integer as an IEEE 754 Standard. Convert 10.5_{10} to Single Precision Format

1. Convert to binary
   (a) $10 = 1010_2$
   (b) $.5 = .10_2$
   (c) $+10.5 = 1010.10_2$ x 2^0
2. Normalise (write with a significant bit 1 before the dot) $1.01010 * 2^3$. The 1. is not stored in the computer; this increases room for precision by one bit. The 1 is put back in arithmetic operations.
3. Exponent is +3 in excess 127 format = 127_{10} + 3_{10}=130_{10} = 1000 0010_2
4. Mantissa = 0101 0000 0000 0000 0000
5. Sign = 0, positive

Table 5.1: IEEE 754 Single Precision

sign	exponent	mantissa
0	1000 0010	0101 0000 0000 0000 0000 000

If we now regroup them in groups of 4 bits:
0100 0001 0010 1000 0000 0000 0000 0000 = $4128 0000

Some remarks:

- In single-precision, when the exponent is 255
  - if the significand is zero, the quantity is +-infinity,
  - the significand is non-zero the quantity is Not a Number (NaN), (usually means error)

- In double-precision, when the exponent is 2047
  - if the significand is zero, the quantity is +-infinity,
  - the significand is non-zero the quantity is Not a Number (NaN), (usually means error)

Student Exercises

Convert the following decimal numbers to IEEE 754 FP Format:

1. 56.125
2. –73.01
3. –65.666

5.3.3 Converting IEEE 754 Floating Point to Decimal: Single Precision

Let's assume that we need to convert $40C4 0000 to decimal.

$40C4 0000 = %0 100 0000 1100 0100 0000 0000 0000 0000

Sign = 0 (positive)

Exponent = 1000 0001 = 129_{10} – 127_{10} = 2

Significant =%1.100010000 (Note that the 1 is put back)

Result = 1.10001 * 2^2 = 110.001_2 = 6.125_{10}

5.3.4 Loss of Accuracy

Again, the loss of accuracy of any floating point representation often results in errors in arithmentic calculations. For example, assume that the computer handles 4 bit decimal numbers. Consider the decimal calculation by hand

$$543.57 + 1.862 = 545.432$$

If we align both numbers

$$0.5435 * 10^3 + 0.1862 * 10^1$$

To add the two numbers we must make the exponents equal:

$0.5435 * 10^3 \rightarrow 0.5435 * 10^3$

$0.1862 * 10^1 \rightarrow 0.0018 * 10^3$

And adding them together, we have:

$$0.5453 * 10^3 = 545.3$$

Since the calculation by hand is 545.432, the difference is significant.

Student exercises

Convert IEEE 32-bit FPF to decimal

1. $C013 0000
2. $BEC0 0000
3. $C3C0 0000

5.4 Alphanumeric Codes

In addition to numbers, computers manipulate alphanumerical characters, that is text characters including letters, digits and special characters. The most common codes in use are ASCII, EBCDIC and Unicode codes. Of these the most often used is the ASCII code.

5.4.1 ASCII Code: American Standard Code for Information Interchange

- 7 bit code used world wide, usually stored in a computer in an 8-bit byte with the most significant bit left as 0[1]
- there are 128 different characters
- 10 digits, 0-9
- 26 uppercase letters

[1]Sometimes an 8-bit version is used, by either keeping the number of 1s even (even parity) or odd (odd parity)

- 26 lowercase letters
- 7 punctuation marks
- 34 control characters
- 26 other characters

	0␣	1␣	2␣	3␣	4␣	5␣	6␣	7␣
␣0	NUL	DLE	*space*	0	@	P	‘	p
␣1	SOH	DC1	!	1	A	Q	a	q
␣2	STX	DC2	"	2	B	R	b	r
␣3	ETX	DC3	#	3	C	S	c	s
␣4	EOT	DC4	$	4	D	T	d	t
␣5	ENQ	NAK	%	5	E	U	e	u
␣6	ACK	SYN	&	6	F	V	f	v
␣7	BEL	ETB	’	7	G	W	g	w
␣8	BS	CAN	(	8	H	X	h	x
␣9	HT	EM	)	9	I	Y	i	y
␣A	LF	SUB	*	:	J	Z	j	z
␣B	VT	ESC	+	;	K	[	k	{
␣C	FF	FS	,	<	L	\	l	|
␣D	CR	GS	-	=	M	]	m	}
␣E	SO	RS	.	>	N	^	n	~
␣F	SI	US	/	?	O	_	o	DEL

Table 5.2: ASCII Table

The order assigned within a code is called the *collating sequence*. For ASCII, we have:

```
A 100 0001
B 100 0010
C 100 0011
D 100 0100

0 011 0000
1 011 0001
2 011 0010
```

Thus characters and text strings can be compared, by comparing their ASCII values:

```
A < D
FRED < FREE
0 < 7
X < e
A < a
```

However *this is not alphabetical order*, since for example capital letters come before lowercase in the collating sequence. For example, A < a, and X < b are both true in the ASCII code, but not in the alphabetical order.

Other Codes

- Extended ASCII (not standard): the original ASCII was extended to 256 symbols to represent characters not present in the ASCII standard. Unfortunately, this extension is not a standard.
- EBCDIC: an 8-bit character code invented by IBM, comprising 256 characters
- Unicode: this is a newer code to cater for many of the world's languages, such as Asian languages which have many symbols. It uses 16 (or more) bits to represent each character. A subset corresponds to ASCII—that is, Unicode extends the ASCII code. The Java programming language uses Unicode to represent characters. The complete set may be found at `www.unicode.org/charts`.

5.4.2 BCD Codes

BCD Binary Coded Decimal numbers are used in systems where only numbers are being entered. Although BCD arithmetic can be performed without the need to convert to binary, many systems perform the conversion to implement the operation so they can use the same circuits. The BCD table is as follows:

```
0= 0000     5= 0101
1= 0001     6= 0110
2= 0010     7= 0111
3= 0011     8= 1000
4= 0100     9= 1001
```

Codes 1010 to 1111 are not used.

Example: 1,295= %0001 0010 1001 0101

To make it more efficient the code may be packed, two symbols per byte.

5.5 Conversion between ASCII and binary

This is a very common conversion. For example, a computer converts ASCII keyboard keys to binary, when arithmetic is to be carried out. That is, the keyboard reads ASCII '1' = 0011 0001, but the program needs the integer 1 = 0000 0001. To convert '1' to binary 1 for arithmetic, you have to subtract '0' = %0011 0000, which leaves %0011 0001 – %0011 0000 = %0000 0001.

Similarly, when printing a binary number such as %0000 1001, (= 9), you have to add %0011 0000 to give %0011 1001 which is ASCII for '9'. Converting multi-digit binary numbers, such as %1110 1110 to the corresponding sequence of ASCII characters '238' is more complex though.

However, converting '238' to its integer equivalent is not too difficult, and it is used very often, since strings such as '238' are entered via the keyboard to input integer numbers. This is what the Java method `Integer.parseInt()` does. Via the keyboard, the characters come one by one in a sequence '2', '3', '8'. To construct the binary number, we pick the characters one by one, multiply by 10 and add them each time as follows:

first character → 2

multiply by 10 → 2 * 10 = 20

add second character → 20 + 3 = 23

multiply by 10 → 23 * 10 = 230

add third character → 230 + 8 = 238

To produce a binary number we may use the same algorithm, but the operations are then binary. Multiplying by 10 in binary is not immediate, so we have to use a little maths:
'2' – '0' = 0011 0011 – 0011 0000 = 0000 0011 = 2_{10}
and then use the algorithm, remembering that multiplying by 10 is the same as multiplying by 8 and by 2, and adding:

1. $2_{10} * 10_{10} = 2_{10} * 8_{10} + 2_{10} * 2_{10} = 00000010 * 1000 + 00000010 * 10 = 00010000 + 00000100 = 00010100$ $(= 20_{10})$
2. $20_{10} + 3_{10} = 23_{10} = 00010100 + 00000011 = \ldots$
3. $23_{10} * 10_{10} = \ldots$

Exercise: Finish this off

Student Exercises

1. Convert the text string `'Yoko's in Tokyo'` to 7-bit ASCII string.

2. Convert the string `'Mary's eyes are blue'` to an 8-bit ASCII string.

3. Convert the 7-bit ASCII code to text:

   1000111110111111011111001000100000

   1101101110111111100101101110110100111011101100111010

   0000100110111100100100000101001111011011101001111010

4. Convert the ASCII 8-bit code to text:

   0100100001101111011011000110100 1

   0110010001100001011110010010111 0

5. Convert the string '137' to its equivalent binary

Practical Work 5

5.6 Tutorial exercises

1. List the steps required to execute the instruction: `d0 = d1 - d2` on your CPU diagram.

2. Draw a memory diagram showing how bytes, words and long-words are stored.

3. Describe the sequence of steps performed by the OS when a data item is not in memory and it must be brought in from disk, including cache and virtual memory aspects.

4. Perform the following multiplications, first unsigned and then signed. Compare your results.

```
1110 0101 * 1101
0110 1100 * 1001
```

5. Convert first the numbers to binary and then perform the following binary operations (8-bits max for fractional part):

```
15.25 + 17.45
4.90 + 34.25
```

   (a) Perform the following multiplications assuming *unsigned data*, verifying your answer by converting both the equation and solution to *decimal* afterwards:

      i. $0010\,1110_2 \times 001_2$

      ii. $0011\,1011_2 \times 011_2$

      iii. $1000\,1011_2 \times 110_2$

   (b) Perform the following multiplications assuming *2's complement data*, verifying your answer by converting both the equation and solution to *decimal* afterwards:

      i. $1110_2 \times 1001_2$

      ii. $1011_2 \times 0111_2$

      iii. $0011_2 \times 1000_2$

(c) Convert the following decimal fractions to *binary fractions* no longer than *8 bits* in length:

i. `0.5`

ii. `0.25`

iii. `0.75`

iv. `0.565`

v. `0.1013`

5.7 Laboratory exercises

1. Create a file with the sample program in Section 6.1, and assemble and run it. Experiment with the emulator by tracing the program.

2. Modify the program above so it allocates a word (not long word) for `data2`, and adds a new data4, also a word. Add `data1` and `data2` and store into `result1`, and `data3` and `data4` and store into `result2`. Subtract `data2` from `data1` and store the result into `result3` (This exercise must be submitted as part of Assignment 1.)

Chapter 6

Computer Organisation (cont.) Assembly Language (cont.)

Objectives

After studying this chapter, students should be able to:

- Explain memory alignment and its consequences
- Explain what exceptions are, and discuss software and I/O interrupts, including the interrupt mask
- Explain what Direct Memory Access and how it helps to make I/O more efficient
- Describe the Status Register and its relationship with branching
- Create programs with branching instructions

6.1 Sample Assembly Program

This sample program adds together three long words at memory locations `$2500`, `$2504`, `$2508` and saves the result in location `$250C`. This program uses labels, which are symbolic references to addresses. We can use the place where a label is defined — the *location counter* — to refer to the address. For example, the following program:

Code Example 6.1: Add 3 numbers – declared memory

```
1               org         $1000
2               move.l      #$7ffe,sp       ;init stack pointer
                move.l      data1,d2        ;get data1
4               add.l       data2,d2        ;add data2
                add.l       data3,d2        ;add data3
6               move.l      d2,result       ;save result
                move.b      #9,d0           ;stop
8               trap        #15

10              org         $2500
    data1       dc.l        1               ;initialised memory = 1
12  data2       dc.l        2               ;initialised memory = 2
    data3       dc.l        5               ;initialised memory = 5
14  result      ds.l        1               ;one long word for the result
                end
```

The above code now allocates four memory locations for the data. The locations `data1`, `data2` and `data3` are long words reserved and initialised by the `dc` instruction, whereas `result` is allocated but not initialised by the `ds` instruction. The labels `data1`, `data2`, `data3` and `result` represent the locations (addresses) of the data; that is, the assembler identifies the locations by the labels, so the program can refer to the address of the first integer as `data1`, for example. When the program is assembled (i.e. converted into the machine language of the CPU) the labels will be replaced with the actual addresses of the storage locations. But, if the locations were moved, the assembler would still put the correct address associated with each label into the machine code. We don't know the addresses of the data — and we should not care — but the computer does. We should always use labels to avoid 'hard wiring' addresses in our code. As an experiment, you could try reordering the lines with data1, data2, and data3, just to see what happens in the assembled program and whether it still runs correctly.

The code starts as before by initialising the stack pointer, and moving the first `data1` item into register `d2` where the sum will be accumulated. Then both `data2` and `data3` are added to `d2`, and finally the sum is placed into `result`. After that, the program exits.

NOTE: spaces (or tabs) before instructions, no spaces before labels.

6.1.1 Memory Alignment Revisited

We have already mentioned in Section 4.4 that certain memory organisations require that objects are located at particular addresses in memory, that is, the processor expects that words or long words start at particular addresses in memory and that this is called *memory alignment*. This is because for a 16-bit bus (32-bit bus) the hardware is often set up to to transfer 16 bits (32 bits) at a time that way. The 68K, for example, requires that all words and long words are stored at even addresses in memory (i.e. LSB is 0). That is called a *word boundary*. If for any reason this does not happen, the 68K processor throws an exception (see next section). In this case we say that the 68K alignment is a *hard alignment*. That is why we use the line:

```
move.l      #$7ffe,sp
```

to initialise the stack pointer on an even address in memory, instead of using the very top of available memory `#$7fff` which is an odd address.

However, some other processors (e.g. Intel 8086) have softer alignment requirements, they accept a misalignment and do not *throw an exception* (*raise an exception*) if data are not aligned. This is called *soft alignment*. However, there is a significant performance penalty since if a word spans a word boundary the processor has to access two data words to access the required word.

6.2 Exceptions

When a processor is executing a program, exceptional behaviour (exceptions) may occur:

External exceptions: Interrupts, interruptions by external devices, for example indicating that new characters are available for the processor to consume;

Program exceptions: typically an operation exception, such as 'attempt to divide by zero' or a floating-point operation that gives overflow; the processor cannot go ahead so the operation must be stopped. Also, *illegal instructions* and *unimplemented instructions* produce this type of exceptions, which are detected by the decoding system.

Instruction/Data access exceptions: There are normally raised by the memory management system when an attempt is made to access an instruction or data that is not physically present, or that is outside of the region(address range) allocated to a process.

1. The first case was presented before as a page fault, issued by the virtual memory system.
2. In the second case, the exception is a response to an attempt to access memory beyond the process allocated area: this is the most common cause of the 'Bus Error' message encountered by programmers.

Software interrupts: Requests by the programmer to the operating system to perform a task, e.g. to write characters to a file on disk. These operations are too delicate and dangerous to leave them to the programmer, so they are not implemented by a normal program instruction. An operating system provides ready-to-run software instructions — *privileged instructions* — and corresponding routines to perform these operations. These *system calls*, traditionally called *traps*, execute in *supervisor mode* (more on this later) to be able to access the hardware.

In all cases, when an interrupt is detected the processor stops execution, and attends to the interrupt as follows[1]:

1. An interrupt number or type is obtained (from the device, the software, etc. This is the vector.)

[1]To be precise, this description corresponds to *vectored interrupts*.

2. This number is used to access a table in the lower section of memory
3. The table entry points to — i.e. contains the address of — the *interrupt routine* (*exception or interrupt handler*)
4. The system produces a context switch by:
   (a) saving the state of the executing process on the system stack;
   (b) dispatching the interrupt routine by initialising the registers with the routine information and loading the Program Counter with the routine start address;
   (c) executing the routine (usually) until the end;
   (d) restoring the original state and then returning to normal program execution.

After executing the exception handler, the program may exit (it was a *fatal exception*) or continues executing. Please note that some exceptions are fatal — e.g. divide by zero — that is, the processor cannot continue executing the program and the exception handler most likely will display an error message and exit, while others are non-fatal and the exception handler may fix up or skip over the error and return control to the running code. It is responsibility of the programmer to deal with non-fatal exceptions properly, and avoid killing the program for every exception.

Please note that the task 4a above is *critical*, that is, the task itself cannot be interrupted since that another interrupt may result in the corruption of the information being saved on the stack. Thus, processors have special instructions to disable interrupts during a critical task such as saving the processor state when performing a context switch.

6.2.1 Programmer Interrupts

In the EASy68K simulator, programmer interrupts (traps) are implemented by initialising (by executing a `move`) `d0` with a function number and then executing the `trap` instruction[2]. The operating system uses the function number provided (see 1 above)

[2]Please note that a0 and a1 may be changed by a trap function. This is implementation-dependent, so it is not possible to anticipate, and it may be the source if inexplicable errors.

to determine which routine to execute. Some useful functions are: [3]

```
#5          ; INCH, reads a keyboard ascii key into d0
#6          ; OUTCH, outputs ascii in d0 to the screen
#9          ; EXIT, exits back to the simulator
```

Note: The character that is read is placed in the `D1.b` register, and the character that is displayed is retrieved from the `D1.b` register.

Example:

```
move.b      #6,d0           ;prepare to display
trap        #15             ;display the character
```

The following program inputs and displays (echoes) a keyboard character to the display. Create and test.

Code Example 6.2: Get and echo keyboard character

```
1       org         $1000
2       move.l      #$7ffe,sp       ;init stack
    ; read the keyboard
4       move.b      #5,d0           ;prepare to read a character
        trap        #15             ;get the character
6   ; display the key
        move.b      #6,d0           ;prepare to display
8       trap        #15             ;display the character
    ; stop
10      move.b      #9,d0           ;prepare to exit
        trap        #15             ;exit
12      end
```

[3] The EASy68K simulator, unlike the previously used Sim68K, also has interrupt functions to read and write strings. These will be discussed at a later time in the course. And, in the meantime, use the single character input and output functions since coding for multiple characters is a useful exercise.

6.3 The Fetch-Decode-Execute Cycle Revisited

Recall from Section 3.1.1 that this cycle represents the steps that the computer follows when running a program:

1. Fetch:
   (a) Copy the PC onto the MAR
   (b) Fetch the instruction in the MAR and place it in the IR
   (c) Increment the PC to point to the next instruction
2. Decode: Interpret the instruction, and if there is a need to fetch data place the address in the MAR
3. Execute:
   (a) Use the address in the MAR to get the data (if required) and place it in the MBR
   (b) Execute the instruction

To avoid the processor having to ask all the time whether there has been some input, and to be able to efficiently service interrupts, the CPU is allowed to be interrupted (say, by a signal from an I/O device) by checking for interrupts at the beginning of each cycle. A program exception may be handled in a slightly different way, since the exception may make impossible to complete the instruction currently executing (say, a 'divide by zero' exception).

If the processor detects an interrupt, it first attends to the interrupt, and then resumes normal execution. The sequence of events when processing an interrupt is described next.

6.3.1 Input/Output, Interrupt Priority and the Interrupt Mask

If an interrupt is issued by an I/O device, the CPU:

1. stops executing after the current instruction;

2. attends to the interrupt;

3. identifies the interrupting device;

4. invokes the interrupt handler, and the sequence proceeds as described before.

However, in I/O the *interrupt priority* is very important. Each interrupt has a priority, and interrupts of lower priority are allowed to be interrupted by those of higher priority. In this way, less urgent interrupts are themselves interrupted by more urgent ones. In I/O, priority depends on the device: devices with shorter *response deadlines*, such as a network connection, should be serviced first, so to minimise the risk of losing data. Therefore, the interrupts associated to a device such as a keyboard are of lower priority than network interrupts, for example.

The above discussion means that an interrupt routine may be interrupted mid-execution, and another routine may be dispatched to run. This creates a situation called *nested interrupts*, where one interrupt routine may be waiting for another to finish, which in turn may be waiting for another, and so on. Recall that the 68K processor has a system byte in the status register to store information to be used by the system. Three bits of this byte — bits 8, 9 and 10 — are the *interrupt mask*, that is, the bits that indicate the priority of the current interrupt, and hence, whether the running routine is to be interrupted by an incoming interrupt. If the mask value is 100 (level 4), it will not be interrupted by interrupts with priorities 000, 001, 010, 011, 100. Only interrupts with priorities greater than 4 — i.e. 101, 110 and 111 — would be serviced. When an interrupt occurs, the mask is changed to the new level; thus, if an interrupt with priority level 5 is being serviced, the mask in the status register is changed to 101.

Notice that during normal operation the processor runs at level 0, i.e. no interrupts. Thus all interrupts should eventually be serviced.

Finally, most processors support *non-maskable interrupts*, that is, interrupts that cannot be ignored and have to serviced immediately. This is to attend to very urgent tasks, such as the computer going down or as a response to some urgent outside world

event. The 68K uses level 7 as a non-maskable interrupt, in the sense that an interrupt level 7 is always serviced by the processor. In the unlikely event in which the processor detects an interrupt level 7 while servicing another interrupt level 7, the new interrupt is serviced.

6.3.2 Direct Memory Access: DMA

Up until now we have assumed that the responsibility for managing and moving data around is concentrated exclusively in the processor. However, with modern fast devices even the fastest CPU will struggle to cope with the speed of I/O transfers. In addition, there would be a waste of CPU cycles employed in moving data around. The solution, in principle, is simple: allow simple programmable device controllers to read from and write to memory directly. Devices with this capability are known as *Direct Memory Access (DMA)* devices.

In order to perform a transfer, the CPU provides the device controller with the location of the bytes to be transferred, the number of bytes to be transferred and the destination device if it is a write or the memory address if it is a read. The CPU then signals to the controller that the information for the transfer is ready, and keeps going with other tasks. From then on the CPU and the DMA device compete for access to the same bus, but only one of them at the time may be the *bus master*. In this operation the DMA becomes the bus master for the period of a read or write, performs the required transfer, and then frees the bus for the CPU or another device to use. In this context, a read or write refers to moving one "block"[4] of the data that needs to be moved. Note that a DMA device interferes with CPU execution since it steals cycles from the processor, but this does not significantly affect performance since there is no context switch, that is, all the while the CPU may continue executing the same program. And there will be times in the running of a program when the CPU does not actually need access to the bus. It is at these times that the cycle stealing occurs.

[4]Block is possibly not the best term for this. When the DMA controller "steals" a cycle from the CPU, it moves a block of data that is equivalent to the width of the data bus. So, on a 68K, it would move 32 bits(4 bytes) of data. Or, with one of the newer 64 bit CPUs in use, the DMA would move 8 bytes with every cycle that it steals.

6.4 I/O Devices

A common way of implementing interrupt-driven I/O in small computer systems is to provide each device with an interrupt request line (and corresponding number). In this way, when a device needs to interrupt the CPU it raises its I/O interrupt line, and then the Interrupt (INT) line is raised in the CPU, usually by a chip controller. When the CPU is ready to process the interrupt, it raises the Interrupt Acknowledge (INTA) line to start the process. Typically there are more devices than available interrupts, so interrupts numbers must be shared. Devices unlikely to clash such as a printer and a scanner may share the same number, but there may be conflicts between others. The I/O controller makes sure that all connected devices work without conflict.

6.4.1 Ports

The contact points of the computer communication with the outside world are the *ports*. These allow the transfer of data between the computer and external devices such as a mouse, a hard disk or a printer. There are two main types of communication strategies, and therefore, two types of ports: serial and parallel. In serial communication bits are transferred one at a time in a sequence, while in parallel communication several bits are transmitted in parallel at the same time. Although parallel communication is in principle faster than serial because it sends many bits at a time, typically it is harder to have error-free parallel communication over long distances due to slight imperfections in the connecting wiring and the loss of signal energy. Thus, parallel connections are restricted to short distances, such as a local printer, for example. In contrast, serial communication does not have this limitation, and may be used over longer distances.

Lately, some standards for fast serial communication have emerged:

- *USB, Universal Serial Bus.* A popular standard bus that supports 'Plug and Play' (the ability to detect and configure devices automatically), and 'Hot Plugging' ('Hot Swapping') (the ability to connect and disconnect devices on a running computer). The USB provides interconnection between PCs and a wide variety of peripherals.

Version 1.1 of this standard supports data transfer rates of 1.2 megabits per second and 12 megabits per second while version 2.0 supports rates up to 480 megabits per second. USB supports bi-directional data transmission, and also includes +5V and ground wires for supplying power to devices such as iPods and portable hard drives.

- *FireWire.* A very fast peripheral standard, developed to connect the computer with multimedia peripherals such as digital video cameras and other high-speed devices like the latest hard disk drives and printers. FireWire ports operate at up to between 400 megabits per second and 800 megabits per second in normal operation. FireWire is an implementation of the high-speed serial data bus defined by the IEEE 1394-1995, IEEE 1394a-2000, and IEEE 1394b standards. It features simplified cabling, hot swapping, and transfer speeds of up to 800 megabits per second (on machines that support 1394b).

6.5 Flow of Control: Program Branches

One of the main features in a computer program resides in being able to make decisions when it runs. For instance, to repeat some operations many times, or to skip over some code according to the result of a logical evaluation. The fundamental flow of control statements are:

Sequence: The statements are executed in sequence, one after the other. This is the normal operation of the processor. See Figure 6.1

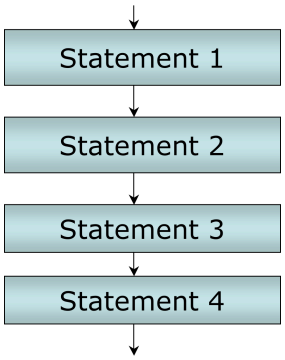

Figure 6.1: Sequential statements

Choice: the equivalent of an `if ...else` statement. A different execution path is taken according to the true/false value of a condition. See Figure 6.2.

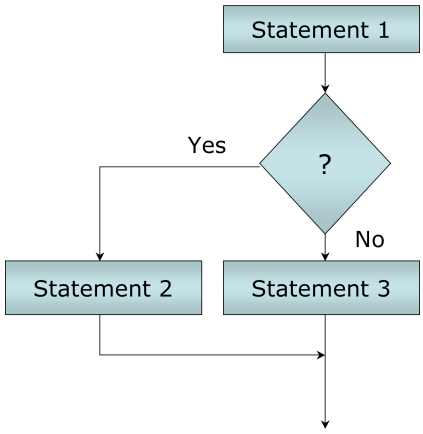

Figure 6.2: A choice: an `if...else`

Loop: these are iterations:

- where the loop is executed at least once. This is equivalent to a `do ...while` loop. Figure 6.3 illustrates.

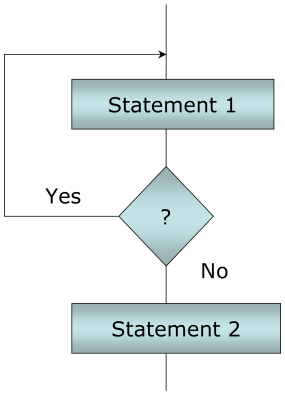

Figure 6.3: A `do...while` loop

- where the loop may not be executed at all. This is equivalent to a `while` loop, illustrated in Figure 6.4.

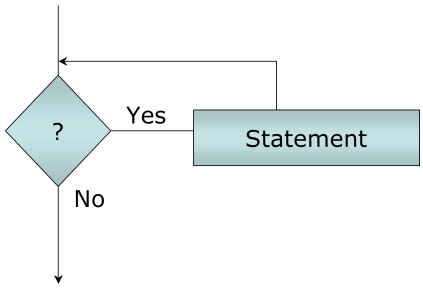

Figure 6.4: A `while` loop

These usually are implemented in a high level language such as Java or C, but, as we will see, we can also implement them in assembly. High-level language loops have an immediate translation into assembly. To see how this works at the processor level, we first have a careful look at the Status Register.

6.6 Status Register

We have already mentioned the Status Register, as a 16-bit register where the lower byte represents the result of operations within the CPU for programmer's use, and the higher byte is used by the system to control the running of programs. The bits for the programmer (recall, the Condition Code Register CCR) are inspected to find out whether the previous instruction resulted in zero, negative, overflow, etc. The higher byte controls some execution parameters such as the interrupt mask discussed before, whether the processor is executing in user or supervisor mode, whether is is tracing (i.e. stopping after each instruction), etc. See Figure 6.5.

Condition Code Register : users' byte

- C = carry
- V = overflow
- Z = zero
- N = negative
- X = extend

System Byte: processors byte

- 15 = trace mode (1=on, 0=off)
- 13 = supervisor mode (1-on, 0=off)
- 8, 9, 10 = interrupt mask

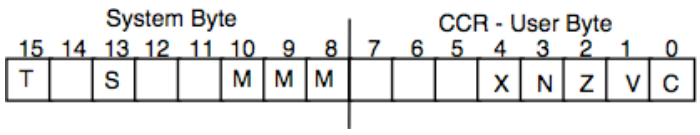

Figure 6.5: Status Register

One very important function of the CCR is related to branching in programs. Almost every operation results in a change of the CCR bits; of particular importance are conditions to resume program execution at a different location if the previous operation had

a zero result and the zero bit has been set, that is, `Z = 1`. For example, if a variable `counter` is initially 3 and it is decremented 3 times, at the end of the process `counter` will be zero and then `Z = 1`. Thus, it is possible to execute a set of statements a fixed number of times and then exit at the end of the loop.

6.6.1 Branch Instructions

Branch instructions allow the programmer to make decisions about the flow of a program, as in the `if...else` or `while` discussed before. There are 14 branch instructions in the 68K. In each case some bits in the CCR are set or reset — e.g , a comparison has been made, or an operation resulted in a zero result — and a decision is made accordingly by the branch instruction:

`beq` Branch If Equal. Take the branch if the result of the previous instruction was zero

`bne` Branch If Not Equal.Branch if the result of the previous instruction was not equal to zero

`bge` Branch If Greater or Equal. Branch if the result of the previous instruction was greater than or equal to zero

`blt` Branch If Less. Branch if the result of the previous instruction was less than zero

`ble` Branch If Less or Equal. Branch if the result of the previous instruction was less than or equal to zero

`bgt` Branch If Greater. Branch if the result of the previous instruction was greater than zero

`bra` Branch Always: Always takes the branch

`bcc` Branch If Carry Bit is Clear (C=0)

`bcs` Branch If Carry Bit is Set (C=1)

Consider the following examples using branch instructions:

if: Assume that we have an integer number in the lower byte of `d1`. The following code implements an `if` in assembly:

```
        cmp.b    #5,d1      ; compare d1 with 5
        bge      skip       ; if d1 >= 5 skip if
        ..........          ; if statements here
        ..........
skip    ..........          ; condition was false
```

The code above implements:

```
 if (d1 < 5)
    // if segment;
```

if...else : Assume that we have an integer number in the lower byte of `d1`. The following code implements an `if...else` in assembly:[5]

```
        cmp.b    #5,d1     ; compare d1 with 5
        blt      if        ; if d1 less than 5 branch to if
        ..........         ; else here
        ..........
        bra       skip     ; skip the if clause
if      ..........         ; if segment
        ..........
        ..........
skip    ..........
        ..........
```

[5]Consider the way that the flow of code jumps around in this statement. Experienced (old) programmers refer to languages that force the use of these constructs as "spaghetti code", refering to the difficulty of tracing the flow of execution.

Ths code on the previous page implements:

```
if (d1 < 5)
   // if segment;
else
   // else segment;
```

do...while and while :

- The following assembly code implements a `do...while` loop.

```
        move.b     #0, d1
next    add.b      #1, d1
        ..........
        ..........
        cmp.b      #7, d1
        blt        next
        ..........
```

  This is equivalent to the `do...while` loop:

```
d1 = 0;
do
{
   .........
   .........
} while (d1 < 7);
```

- The following assembly code implements a `while` loop.

```
        move.b     #0, d1
next    cmp.b      #7, d1
        bge        follow
        ..........
        ..........
        add.b      #1, d1
        next
follow  .........
```

This is equivalent to the `while` loop:

```
d1 = 0;
while (d1 < 7)
{
   .........;
   .........;
}
```

Please note that although in high level languages it is common to write loops with an increasing counter — e.g. `for (i = 0; i < 5; i++)` — given that the branch instructions are often related to zero, loops in assembly are very often implemented with a decreasing counter, say from 4 down to 0.

These branch instructions in the 68K are usually used together with an extra addressing mode that makes things easier for the programmer, as we shall see in the next chapter.

6.6.2 Using Branch Instructions

Branch instructions are used to branch to a different section of code if a certain condition is met, or not met. The following code may be used to detect whether a character in a register — `d0` — is a digit, and process it accordingly (assuming the existence of the two routines `yes_digit` and `no_digit` :

```
is_digit        cmp.b   #'0',d0     ; every character with an ASCII less
                blt     no_digit    ; than '0' is not a digit
                cmp.b   #'9',d0
                ble     yes_digit   ; everything >= '0' and <= '9' is
                bra     no_digit    ; everything > '9' isn't
```

Of course, branch instructions may be combined to implement multiway decisions. The following code determines whether a character in `d0` is a word character (either a digit or a letter character):

```
is_word_char    cmp.b   #'0',d0     ; characters with an ASCII value less
                blt     iwc_no      ; than '0' are not a word character
                cmp.b   #'9',d0
                ble     iwc_yes     ; everything >= '0' and <= '9' is
                cmp.b   #'A',d0
                blt     iwc_no      ; everything > '9' and < 'A' isn't
                cmp.b   #'Z',d0
                ble     iwc_yes     ; everything >= 'A' and <= 'Z' is
                cmp.b   #'a',d0
                blt     iwc_no      ; everything > 'Z' and < 'a' isn't
                cmp.b   #'z',d0
                blt     iwc_yes     ; everything >= 'a' and <= 'z' is
                bra     iwc_no      ; (everything > 'z' isn't)
```

Please note that the `is_digit` code above may be written in a different, somewhat shorter, way:

```
is_digit        cmp.b   #'0',d0      ; every character with an ASCII less
                blt     no_digit    ; than that of '0' is not a digit
                cmp.b   #'9',d0
                bgt     no_digit    ; everything > '9' is not a digit
```

We chose the first implementation because it fitted better with the `is_word_char` code segment.

Practical Work 6

6.7 Tutorial exercises

1. What are the three fields in the *IEEE 754 Single Precision* format? What information do they store and what are their sizes?

2. Convert the following decimal numbers to IEEE 754 Single Precision format, giving your final answer as both *binary* and *hexadecimal* strings:

   (a) `-9832.125`

   (b) `1046.3`

   (c) `2147.98`

3. Convert the following IEEE 754 Single Precision format numbers to their decimal equivalents:

   (a) `$ BD18 0000`

   (b) `$ C2F1 0000`

4. Convert the decimal number `-9832.125` to IEEE 754 Double Precision format.

5. Convert the following string into 7-bit ASCII, giving your final answer as a hexadecimal string:

   `Help!`

6. Using *even parity*, convert the same string into *8-bit* ASCII, giving your final answer as a hexadecimal string.

7. Write an assembly language program that performs the following sequence of actions:

   - store the *byte* with *decimal* value `15` in register `d3`;
   - store the *byte* with value `$ 15` in register `d4`;
   - clear a *word* of memory at address `$ 2000`; and then
   - set the contents of memory address `$ 2000` equal to the sum of the bytes of data stored in registers `d3` and `d4`.

8. Write an assembly program that will prompt the user to enter two digits under 5 and display the sum of the two digits on the screen (i.e. if the user enters the numbers `3` and `4`, the number `7` will be displayed on the screen).

   - What must you remember to do before and after the addition?
   - This program may at first appear unresponsive to the user. Why is this and what can be done about it?

9. Modify the program above so that it echoes the keyboard input to the screen as it is typed by the user.

10. Explain the principle of memory alignment, including hard and soft alignment.

11. List the possible exceptions that may occur during program execution, and explain each in a paragraph.

12. How is an interrupt dealt with by the system? Explain.

13. Explain what the interrupt mask is and how it relates to the handling of I/O interrupts.

14. Describe the difference between cycle stealing and context switch.

6.8 Laboratory exercises

Code some of the sample programs seen during the lecture, and assemble and run them. Experiment with the simulator by tracing through the programs.

Chapter 7

Assembly Language (cont.) Introduction to Digital Logic

Objectives

After studying this module, students should be able to:

- Describe more assembly addressing modes and their relationship with loops
- Present the principles of Boolean Algebra, and its operators
- Simplify Boolean expressions using Boolean Algebra rules
- Describe the basic logic gates

7.1 Loops: Address Register Indirect Mode

Although it is possible to implement loops with the instructions we have seen already, to better make use of loops another 68K addressing mode is convenient: `(an) Address Register Indirect`. With `(an)` the CPU uses the specified register to supply the address of the data in memory, as in:

```
move.b  (a3),d0
```

Rather than the contents of `a3` being the data to be accessed, the content of `a3` is the address of the data to be accessed. In this way, by putting this instruction in a loop and then incrementing the content of `a3` — that is, incrementing the address of the data — on each iteration of the loop, the CPU will access incremental locations each

time around the loop. In this way, a register such as `a3` may be used to sequentially access all the characters in a string, for example.

Sample Program

The flow of control structures discussed in the previous section make possible to create a flow chart to illustrate the logic of a program. For example, Figure 7.1 shows the flowchart corresponding to the program below.

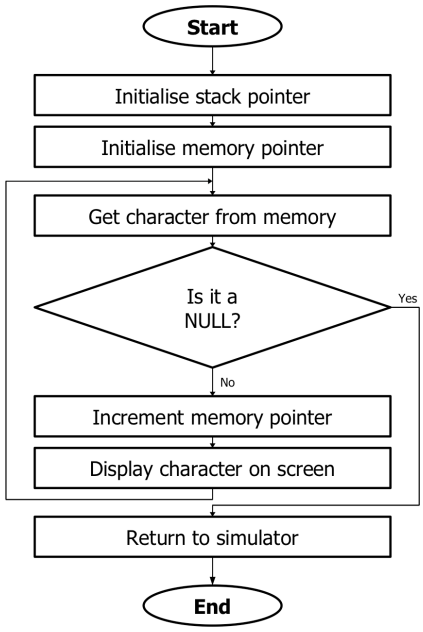

Figure 7.1: Flowchart for a program to display a string

Code Example 7.1: Print a null terminated string

```
1   ; prints a string stored in memory using a loop
2
            org         $1000           ;code space
4           move.l      #$7ffe,sp       ;always the first instruction
            move.l      #string,a3      ;initialises a3 = $2000
6   next    move.b      (a3),d1         ;get character from string
            beq         exit            ;terminate when a 'null' is found
8           add.l       #1,a3           ;increment a3 to point to next char
            move.b      #6,d0           ;display the char in d1
10          trap        #15             ;display it
            bra         next            ;loop until null character
12  exit    move.b      #9,d0           ;prepare to return
            trap        #15             ;exit back
14
    data    org         $2000           ;data space
16  string  dc.b        'this is the beginning,',cr,lf
            dc.b        'and this is the rest',cr,lf,null
18  lf      equ         $0a             ;define a line feed
    cr      equ         $0d             ;define a carriage return
20  null    equ         00              ;define a null
            end         $1000
```

7.2 High-Level Control Structures in Assembly

The previous example shows how to implement a loop that executes a set of instructions until a certain condition (the end of the string) is true. Consider the following segment:

```
next    move.b     (a3),d1         ;get character from string
        beq        exit            ;terminate when a 'null' is found
        add.l      #1,a3           ;increment a3 to point to next char
        move.b     #6,d0           ;display the char in d1
        trap       #15             ;display it
        bra        next            ;loop until null character
```

This is the assembly version of a `while` loop:

```
while (not the end of the string)
{
   get character;
   display character;
}
```

Other high-level loops may also be translated to assembly in a very natural manner using branch instructions. Consider the following examples:

```
for (i = 1; i < 5 ; i++)
{
   statements;
}
```

This loop may be implemented in assembly as follows:

```
        move.b     #4,d1       ; set up control counter -- not a good idea if I/O
next    beq        exit        ; terminate when counter is zero
        ...........            ; statements here ...
        ...........            ; statements here ...
        sub.b      #1,d1       ; decrement counter
        bra        next        ; loop next
exit
```

Be careful though, because the value of `d1` is just the complement of the value of `i` in the loop. That is, when `i = 1`, `d1 = 4`, when `i = 2`, `d1 = 3`, etc.

Alternatively, the loop:

```
for (i = 0; i < 5 ; i++)
{
   statements;
}
```

may be implemented as follows:

```
        move.b     #5,d1           ; set up control counter i
next    sub.b      #1,d1           ; decrement counter
        ...........                ; statements here ...
        ...........                ; statements here ...
        beq        exit            ; terminate when counter is zero
        bra        next            ; loop next
exit
```

Again, be careful because the `for` loop `i` executes from `0` to `4`, whereas `d1` executes from `4` to `0`. Here is another version:

```
        move.b     #4,d1           ; set up control counter i
next    ...........                ; statements here ...
        ...........                ; statements here ...
        beq        exit            ; terminate when counter is zero
        sub.b      #1,d1           ; decrement counter
        bra        next            ; loop next
exit
```

And another loop, similar but not the same:

```
        move.b     #5,d1           ; set up control counter i
next    sub.b      #1,d1           ; decrement counter
        beq        exit            ; terminate when counter is zero

        ...........                ; statements here ...
        ...........                ; statements here ...
        bra        next            ; loop next
exit
```

Please note that although all these versions are pretty similar, they are not equivalent. For example, what happens if the loop is now changed to:

```
for (i = 1; i < x ; i++)
   statements;
```

and x takes on the value 0, 1, or 2? With x = 0, the first loop is now:

```
        move.b     #-1,d1          ; set up control counter as x-1
next    beq        exit            ; terminate when counter is zero
        ...........                ; statements here ...
        ...........                ; statements here ...
        sub.b      #1,d1           ; decrement counter
        bra        next            ; loop next
exit
```

This will not work because the loop control `d1` never reaches the value zero, and so the loop will never terminate! With x = 1, the first loop is now:

```
        move.b     #0,d1          ; set up control counter as x-1
next    beq        exit           ; terminate when counter is zero
        ...........               ; statements here ...
        ...........               ; statements here ...
        sub.b      #1,d1          ; decrement counter
        bra        next           ; loop next
exit
```

The loop now terminates properly, since `d1` is 0 at the start, and the loop exits immediately as it should. The second version with x = 0 is:

```
        move.b     #0,d1          ; set up control counter x
next    sub.b      #1,d1          ; decrement counter
        beq        exit           ; terminate when counter is zero

        ...........               ; statements here ...
        ...........               ; statements here ...
        bra        next           ; loop next
exit
```

Again, this does not work properly because in the first iteration `d1` is –1 and it will 'never' be zero.

Exercise: discuss the remaining cases, and write a code segment that will be *exactly* equivalent to

```
for (i = 1; i < x ; i++)
   statements;
```

7.3 Introduction to Digital Logic

7.3.1 Boolean Algebra

Boolean algebra is appropriate for computing, since it is based on the notion of true or false, which a computer can easily handle. It consists of operands that are either true or false, and operations that combine those operands. Usually, true is represented as 1, and false as 0. The following shows the most common operators:

OR operator: returns true when either operand is true, as shown in Table 7.1.

Table 7.1: OR operator

OR	+
0 + 0	= 0
0 + 1	= 1
1 + 0	= 1
1 + 1	= 1

AND operator: returns true when both operands are true. See Table 7.2.

Table 7.2: AND operator

AND	.
0 . 0	= 0
0 . 1	= 0
1 . 0	= 0
1 . 1	= 1

NOT operator: returns the opposite value; that is, it returns true if the operand is false, and false if the operand is true. Table 7.3 illustrates.

Table 7.3: NOT operator

NOT	–
0	= 1
1	= 0

7.3.2 Operator Precedence

Similarly to normal arithmetic, Boolean Algebra has an order of operator precedence to make sure that operations are performed in the right order. For example, we know that in normal arithmetic `2 + 3 * 4` must be interpreted as `2 + (3 * 4) = 14`, rather than `(2 + 3) * 4 = 20`, since the multiplication 'binds stronger' than the sum. If we need to change the precedence of the evaluation, we use brackets.

When one of these Boolean operators has a higher precedence than another, it also means that it binds stronger, with similar consequences. As it may be seen in Table 7.4, the highest precedence is the `()`, followed by `NOT`, `AND` and `OR`.

Table 7.4: Operator precedence

()	highest
NOT	
AND	
OR	lowest

Therefore:

$$-\mathtt{a} + \mathtt{b.c} + \mathtt{d} = (-\mathtt{a}) + (\mathtt{b.c}) + \mathtt{d}$$

Note that since the `()` have higher precedence that any other operator, you can use `()` to change the precedence of evaluation to whatever is required. For example the expression above may be changed to:

$$(-\mathtt{a}) + \mathtt{b}.(\mathtt{c} + \mathtt{d})$$

if so desired, the brackets forcing the evaluation the new way.

NOTE: often the notations $\bar{\mathtt{a}}$ or `/a` are used instead of `-a`.

These Boolean operations have particular characteristics that are different to the ones we are used to. The following Table 7.5 shows a few:

Table 7.5: Boolean operators properties

0 + X = X	0 . X = 0	X = $\overline{\overline{X}}$
1 + X = 1	1 . X = X	
X + X = X	X . X = X	
X + $\overline{X}$ = 1	X . $\overline{X}$ = 0	

Since these operators work on binary data, it is often possible to prove an identity such as the ones above by going through all the possible situations that may occur, verifying that the equality holds true at all times. This is called a *truth table* and it is used very often. As an example, let us prove the identity X + $\overline{X}$ = 1 using a truth table:

Table 7.6: A truth table for identity X + $\overline{X}$ = 1

X	$\overline{X}$	X + $\overline{X}$
0	1	1
1	0	1

7.3.3 Multiple Variable Rules

Aside from the single variable identities that we have just seen, there are other useful identities involving more than one variable. Some are given in Table 7.7:

Table 7.7: Multi-Variable Rules

Commutative Law	A + B = B + A
Associative Law	A + (B + C) = A + B + C = (A + B) + C A (BC) = ABC = (AB) C
Distributive Law	A (B + C) = AB + AC
Others	A + AB = A A (A + B) = A A + BC = (A + B)(A + C) A + $\overline{A}$B = A + B AB + $\overline{A}$B = B

In addition to the identities above, the following two identities are very useful for the manipulation of Boolean? expressions. They are called the *de Morgan's Theorems*:

- $\overline{\texttt{A + B}} = \overline{\texttt{A}} \texttt{ . } \overline{\texttt{B}}$
- $\overline{\texttt{A . B}} = \overline{\texttt{A}} + \overline{\texttt{B}}$

NOTE: It is very important to note that
$\overline{\texttt{A + B}} \neq \overline{\texttt{A}} + \overline{\texttt{B}}$
and that
$\overline{\texttt{A.B}} \neq \overline{\texttt{A}} \texttt{ . } \overline{\texttt{B}}$

In fact, intuitively, the de Morgan's Theorems actually say:

"The NOT of an OR is the AND of the NOTs",

and

"The NOT of an AND is the OR of the NOTs"

This is simple logic. Consider for example the second Theorem and reason as follows:
"If and A AND B is false, then either A or B is false, and therefore $\overline{\texttt{A}}$ OR $\overline{\texttt{B}}$ is true"
Conversely,
"If either A or B is false, then A AND B cannot be true, and therefore A AND B is false"

7.3.4 Simplification of Boolean expressions

Given a Boolean expression, it may be very important to simplify it as much as possible, since many of these expressions are to be implemented in digital logic — i.e. with digital components — and the simpler the expression the easier and cheaper the implementation will be. To see how these rules may be used for simplification of expressions, let us see a simplification example (comments follow the style of 68K assembly):

$$
\begin{aligned}
Z &= A.B.C + A.\overline{B}.(\overline{\overline{A}.\overline{C}}) && \text{; de Morgan's theorem} \\
&= A.B.C + A.\overline{B}.(\overline{\overline{A}} + \overline{\overline{C}}) && \text{; } A = \overline{\overline{A}} \\
&= A.B.C + A.\overline{B}.(A + C) \\
&= A.B.C + A.\overline{B}.A + A.\overline{B}.C && \text{; multiply out} \\
&= A.B.C + A.\overline{B} + A.\overline{B}.C && \text{; since } A.A = A \\
&= A.C(B + \overline{B}) + A.\overline{B} && \text{; common factor } AC, B + \overline{B} = 1 \\
&= A.C + A.\overline{B}
\end{aligned}
$$

Simplification example

Trace the operations of the following example, and make sure that you can relate each operation to the rules above.

$$
\begin{aligned}
Z &= \overline{(\overline{A}.\overline{B} + C)} + (B.C) \\
&= (\overline{\overline{A}.\overline{B}}.\overline{C}) + B.C \\
&= (\overline{\overline{A}} + \overline{\overline{B}}).\overline{C} + BC \\
&= (A + B).\overline{C} + B.C \\
&= A.\overline{C} + B.\overline{C} + B.C \\
&= A.\overline{C} + B.(\overline{C} + C) \\
&= A.\overline{C} + B
\end{aligned}
$$

7.3.5 Sum of Products and Product of Sums

When manipulating an expression, it is often useful to leave the expression in a standard form:

$$
\begin{aligned}
&= AB + \overline{A}C + \overline{C} && \text{is a sum of products (SOP)} \\
&= (A + B)(\overline{C} + \overline{B}) && \text{is a product of sums (POS)}
\end{aligned}
$$

Both forms are equivalent, but it is quite likely that for special purposes you will be required to work on an expression and leave the final version in one of the standard forms, most likely SOP.

Student exercises

Simplify the following expressions, the final expression should be in SOP.

1. $\overline{A}.\overline{B}.\overline{C} + A.\overline{B}.\overline{C}$
2. $\overline{(\overline{A}.\overline{B}.\overline{C})} + \overline{A.B.C}$
3. $\overline{a}.b + a.\overline{b} + a.b$
4. $\overline{A}.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C + A.\overline{B}.C$

Determine POS forms of the simplified expressions above

7.4 Digital Logic Elements

7.4.1 Logic Gates

Computer hardware is comprised of binary logic systems, which are made using logic gates. There are three basic logic gates: AND, OR, NOT and the derived negated gates: NAND, NOR, XOR, XNOR. All logic systems may be built out of the first three, or out of NAND and NOR gates. These gates are implemented in standard configurations, and later on these in turn are combined to produce complete circuits.

AND gate: There are two inputs and one output. The output is true only when both inputs are true: when all inputs are 1, the output is also a 1.

The AND Boolean expression is: `A.B`. It is represented by a diagram such as the one in Figure 7.2.

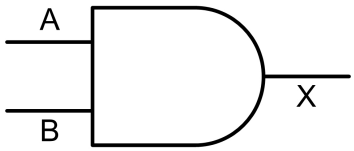

Figure 7.2: AND gate

The behaviour of a gate may be expressed using a truth table. This defines the output for all possible combinations of input, as shown in Table 7.8 for the AND gate.

Table 7.8: A truth table for `A . B`

	Inputs		Output
Row	A	B	A . B
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

OR gate: There are two inputs and one output. The output of the gate is true when either of the inputs is true. The equivalent Boolean expression is `A + B`. The corresponding symbol is shown in Figure 7.3:

The truth table corresponding to the OR gate is depicted in Table 7.9

NOT gate: There is a single input and a single output. The output is the opposite state to the input, i.e. the output is the input inverted. The symbol to represent a NOT gate is shown in Figure 7.4, and the corresponding truth table is Table 7.10.

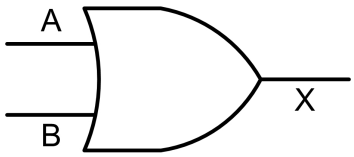

Figure 7.3: OR gate

Table 7.9: OR gate truth table

	Inputs		Output
Row	A	B	A + B
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	1

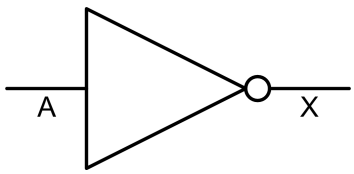

Figure 7.4: NOT gate

Table 7.10: NOT gate truth table

	Input	Output
Row	A	$\overline{A}$
0	0	1
1	1	0

7.5 Combinational Logic

Combinational logic consists of a logic circuit built with logic gates, to implement an output determined by the logic gates and the current set of inputs. There is no memory in the system, that is, the output depends exclusively of the inputs. An example combining and AND gate with an OR gate is shown in Figure 7.5.

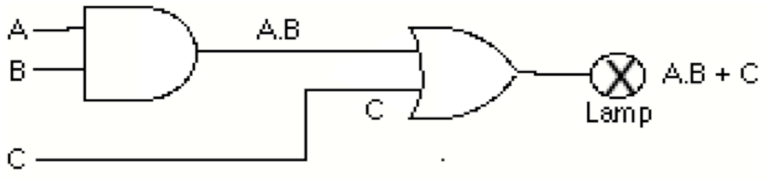

Figure 7.5: An example of combinational logic

The most common situation is to build a circuit from a given Boolean expression. Recall that this is the reason why we try to simplify expressions, so circuits are easier to build. For example, the SOP expression `A.B + C.D` is built very simply with two AND gates joined together with an OR gate, as shown in Figure 7.6

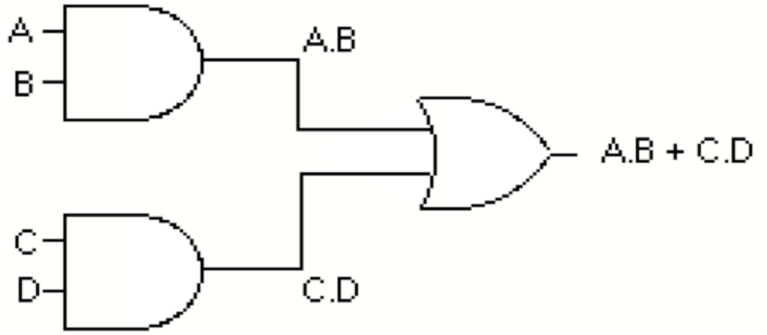

Figure 7.6: A simple combinational circuit

Student Exercises:
Draw logic circuits for the following Boolean expressions:

1. `A.B + C̄ + D`
2. `A + B.C.D`
3. `A.B + C̄.D̄`

7.5.1 Universal Logic Gates

The universal logic gates, NAND and NOR are used to build logic systems instead of the basic AND, OR, NOT as they are more flexible in that *only one type of gate* is needed to build a logic system, and this results in cheaper costs.

NAND gate: It is equivalent to the Boolean Expression: X = $\overline{\texttt{A.B}}$ (NOT(A AND B). The symbol for the NAND gate is shown in Figure 7.7:

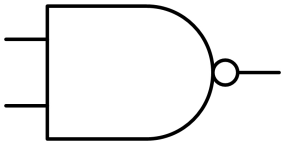

Figure 7.7: The NAND gate

Another notation, also widely used, is shown in Figure 7.8:

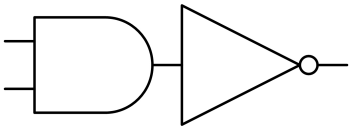

Figure 7.8: Another symbol for a NAND gate

The truth table associated with the NAND gate is shown in Table 7.11:

Table 7.11: NAND gate truth table

	Inputs		Output
Row	A	B	$\overline{\texttt{A.B}}$
0	0	0	1
1	0	1	1
2	1	0	1
3	1	1	0

NOR gate: It is equivalent to the Boolean expression X = $\overline{\texttt{A + B}}$ (NOT(A OR B).

The symbol for the NOR gate is shown in Figure 7.9:

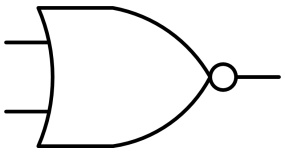

Figure 7.9: The NOR gate

An equivalent symbol is shown in Figure 7.10:

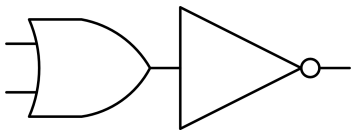

Figure 7.10: Another symbol for a NOR gate

The truth table associated with the NOR gate is shown in Table 7.12

Table 7.12: NOR gate truth table

	Inputs		Output
Row	A	B	$\overline{\texttt{A + B}}$
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	0

XOR gate: This is a slightly different gate, very commonly used in digital logic. It is called the Exclusive OR, written with the expression A $\oplus$ B. For this gate the output is true when the inputs are different, and it is false when the inputs are the same. It is represented by the symbol in Figure 7.11:

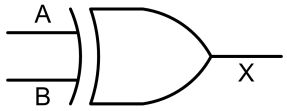

Figure 7.11: The XOR gate

The truth table associated with the XOR gate is shown in Table 7.13

Table 7.13: XOR gate truth table

	Inputs		Output
Row	A	B	A $\oplus$ B
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

XNOR gate: This gate is the negation of the XOR gate, and so it corresponds to the Boolean expression $\overline{\texttt{A} \oplus \texttt{B}}$. Naturally, the output of the XNOR gate is 1 when both inputs are the same, as this is opposite of the XOR gate. The symbol is depicted in Figure 7.12

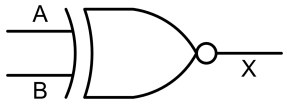

Figure 7.12: The XNOR gate

The truth table associated with the XNOR gate is Table 7.14:

Table 7.14: XNOR gate truth table

	Inputs		Output
Row	A	B	$\overline{\texttt{A} \oplus \texttt{B}}$
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	1

Exercise:

Draw these logic circuits using the gates indicated, and verify them using EasySim:

1. `A XOR B` = $\overline{\texttt{A}}.\texttt{B} + \texttt{A}.\overline{\texttt{B}}$
2. `A XNOR B` = $\overline{\overline{\texttt{A}}.\texttt{B} + \texttt{A}.\overline{\texttt{B}}}$

Practical Work 7

7.6 Tutorial exercises

1. Modify the program in Practical Work 6 to test whether the integers are effectively less than `5`, and to exit the program if they are not.
   - Which control structure must you use for this?
   - What is its equivalent in Java or C?
2. Create and test the program in Section 7.1 in the lab.
3. Use a loop to write a program that produces the three line display, below:

```
One
Two
Three
```

4. Modify the program of Practical Work 6 so that the user has to re-enter the input if an input error is made rather than the program exiting.
   - Which control structure is being used now?
5. Modify the program so that it displays the prompts '`Enter first digit:`' or '`Enter second digit:`' where appropriate before collecting input from the user.
   - How can these strings be stored in memory?
   - Which addressing mode must be used to display them?
6. Discuss the differences between the loops presented in Section 7.2.

7.7 Laboratory exercises

Edit, assemble, run and test the program discussed during the tutorial.

Chapter 8

Error Correcting Codes Introduction to Digital Logic (cont.)

Objectives

After studying this chapter, students should be able to:

- Detect and correct 1-bit transmission errors, using simple parity and Hamming codes
- Design and simplify simple digital systems
- Describe some standard digital components, and their functionality

8.1 Data Representation: Error Correcting Codes

8.1.1 Error Detection and Correction

When transferring data within a computer, for example when writing data to disk, or from one computer to another using a communication line, errors can occur due to noisy lines or simply malfunction. These errors may cause some 1s to become 0s and some 0s to become 1s, as in `"ABC" = 1000001 1000010 1000011`, when transmitted to another computer or stored may become `1000001 1000010 1001011 = "ABK"`. In data communications if an error is detected it may be fixed by requesting retransmission, but when writing data to disk that this may be more difficult.

To solve that problem, extra *redundant bits* (also called *check bits*) are added to the original information before transmission. As a first example, let's add an extra *parity bit* to a 7-bit ASCII character to detect an error. Depending on the number of 1s that we choose, even or odd parity may be used; there are no advantages of one over the other, so they are used indistinctly.

Even Parity: for ASCII characters bit 7 (that is, the eight-bit, the MSB) is set so that the total number of 1s in a character is even, as in:

```
'A' = 100 0001 Parity = 0 Code = 0100 0001
Parity Bit = 0

'C'= 100 0011 Parity = 1 Code = 1100 0011
Parity Bit = 1
```

Odd Parity: the parity bit is set so that the total number of 1s is odd, as in:

```
'A' = 100 0001 Parity = 1 Code = 1100 0001
Parity bit = 1
```

If we consider for example 'A' with even parity 0100 0001, any 1-bit error will change the number of 1s and 0s in the symbol, and thus, the parity of the symbol. Say the error is on bit 3, so the symbol received is 0100 1001; this error is easily detected because the number of 1s is 3 (odd) instead of an even number.

NOTE: A single parity bit can only *detect* an error, it cannot be used to *correct* errors; we need more redundant bits if we want to also correct errors.

Exercises:

- Find Even Parity for: `A, b, c, D, E, f, G`
- Find Odd Parity for: `A, 3, 7, F, Q, q, ?`

8.1.2 Hamming Codes

Simple parity is a particular example of an *error detecting code*. We have just mentioned that although using simple parity it is possible to detect errors, it is not possible to correct them.

Say we want to send a message: Yes = 1, No = 0. If there is an error in transmission, the receiver wouldn't know because they would not be able to detect a 0 changed into a 1, or a 1 into a 0. If instead we use an extra parity bit, and we agree that the number of 0s must be even, for example, to send a 0 we send 00 and to send a 1 we send 11. If there is a one-bit error, the receiver will get 01 or 10, and they will know that there has been an error. However, they will not be able to find out what was the original message.

If instead we use 3 bits to send our message, our messages are always either No = 000 and Yes = 111. If we assume that there is only a one-bit error and we receive 001, what symbol was originally sent? Obviously 000, because 111 has more than one bit difference with 001, so that cannot be the source of a one-bit error. Thus, since there are 3 bits difference between 000 and 111, we can find the mistake and correct it.

An *error correcting code* is able to detect and correct bit errors in a (binary) code. In 7-bit ASCII code, a 1-bit change to a valid symbol gives another valid symbol. It is then not possible to detect or correct a one-bit error, because a 1-bit error is still a legal symbol. Adding a parity bit made it possible to detect, but not correct, an error. To detect or correct errors, codes use extra bits to increase the distance between valid symbols, as in:

```
1111 1000 -> 111 1001 -> 1111 1011 -> 1111 1111
  valid      invalid      invalid       valid
```

In a code, the number of bit changes between two valid symbols of the code is called *Hamming Distance*. In the example, the Hamming Distance has been made constantly 3, and therefore 3 bits must change to arrive at the next valid symbol. Thus, if one bit error is detected, it may be corrected by selecting the closest valid symbol.

To detect up to K bit errors, the (minimum) Hamming distance between two valid code words must be K+1. For example, simple parity makes 2 the distance between symbols and we have seen that it detects a one-bit error in ASCII. To correct up to K bit errors, the Hamming distance between two valid code words must be $\geq$ 2K+1. Hence, to detect and correct 1 error requires a Hamming distance of 3.

In the general case, the way we add those bits is up the scheme used. The most popular one is the Hamming code. In the Hamming code the number of Parity/Check bits p must be:

$$2^p \geq m + p + 1$$

where m = number of data bits and p = number of check bits.

Example:

Find the number of check bits required to detect and correct a single bit error in the BCD code for $9_{10} = 1001_2$. As a guess we can try p = 2 then $2^p = 2^2 = 4$, but since 3 + 3 + 1 = 7 and 4 is not $\geq$ 7, p = 2 is not enough. However, if we now try p = 3 then $2^p = 2^3 = 8$ and m + p + 1 = 4 + 3 + 1 = 8. Therefore p = 3 is sufficient.

A Hamming code includes several parity bits, to be able to detect and correct 1-bit errors. In principle the parity bits may go in many different places, but the most common use is interspersed with the data bits, because:

- to be able to put them at the beginning or the end we have to know in advance the number of data bits, and that it not always possible;
- giving them fixed positions means that we know exactly where the extra bits are.

To get the position of the check bits we count the bits from right to left (i.e. LSB to MSB) starting from 1, and the check bits are at positions 1, 2, 4, 8, 16, 32, etc, all powers of 2. All the other bits are data bits, part of the message.

Table 8.1: Example of Hamming code for 6 data bits, P = parity bit, D = data bit

Bit position	10	9	8	7	6	5	4	3	2	1
Data/Parity	D6	D5	P4	D4	D3	D2	P3	D1	P2	P1

The scheme works by using each check bit as a parity bit for a specific group of bits, as follows:

- P1 covers bits 1, 3, 5, 7, 9, 11, 13, 15, etc.

  P1 covers positions where there is a 1 in the first binary bit of the position number binary expression (highlighted in boldface): 000**1**, 001**1**, 010**1**, ...

- P2 covers bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19, etc.

  P2 covers positions where there is 1 in the second binary bit of the position number binary expression: 00**1**0, 00**1**1, 01**1**0, 00**1**1, ...

- P3 covers bits 4, 5, 6, 7, 12, 13, 14, 15, etc.

  P3 covers positions where there is a 1 in the third binary bit of the position number binary expression: 0**1**00, 0**1**01, 0**1**10, 0**1**11, ...

- P4 covers bits 8, 9, 10, 11, 12, 13, 14, 15, etc.

  P4 covers positions where there is a 1 in the fourth binary bit of the position number expression: **1**000, **1**001, **1**010, **1**011, **1**100, **1**101, **1**110, **1**111

We can think about it in a different way:

- P1 covers bits 1, 3, 5, 7, 9, 11, 13, 15, etc.

  P1 covers one bit, skips one bit

- P2 covers bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19, etc.

  P2 covers two bits, skips two bits

- P3 covers bits 4, 5, 6, 7, 12, 13, 14, 15, etc.

  P3 covers four bits, skips four bits

- P4 covers bits 8, 9, 10, 11, 12, 13, 14, 15, etc.

  P4 covers eight bits, skips eight bits

Example: determine the single bit error-correcting Hamming code for the data 1011_2. Use EVEN parity. From the formula we can see that p = 3 is sufficient, so we can write:

7	6	5	4	3	2	1
1	0	1		1		

- P1 checks bits 1, 3, 5, 7 = ? 1 1 1, P1=1
- P2 checks bits 2, 3, 6, 7 = ? 1 0 1, P2=0
- P3 checks bits 4, 5, 6, 7 = ? 1 0 1, P3=0

Therefore the final code is 1 0 1 0 1 0 1.

The idea is that the parity bits cover each bit (data and parity) with a unique pattern, that is each bit (including the parity bits) is covered by a unique combination of parity bits. For example, bit 5 is covered by bits P1 and P3 only and exclusively, and bit 4 is covered by P3 exclusively.

Then, if we check and find a parity error on bits P1 and P3, but no other parity errors, we know that the problem is with bit 5. So, if bit 5 is a 1, we change it to 0, and if it is a 0 we change to 1 to correct the symbol.

Exercise:

1. Determine the single bit error/correction Hamming code for the ASCII character 'J'. Use ODD parity.
2. Detecting and correcting a single bit error. EVEN parity. Find and correct the error in the Hamming 4-bit code, 1000100. Write down the corrected data.

7	6	5	4	3	2	1
D4	D3	D2	P3	D1	P2	P1
1	0	0	0	1	0	0

- P1 checks 1, 3, 5, 7; 0 1 0 1 = 0 no error
- P2 checks 2, 3, 6, 7; 0 1 0 1 = 0 no error
- P3 checks 4, 5, 6, 7; 0 0 0 1 = 1 error

Therefore there is an error in bit number 4, it should be a 1. Thus corrected code is 1001100 and the correct data is 1001.

It may be surprising that we are only interested in 1-bit errors, but in computer operation 2-bit errors are very, very unlikely. If the probability of a 1-bit error is of the order of 10^{-9}, that is 1/1,000,000,000, this is really small, but if a computer makes 10,000,000 moves a second, on average you get an error every 100 seconds = less than 2 minutes. However, since these communications are most likely parallel, 2-bit errors occur when there is one error on two lines at the same time. That is, the probability is the product of the two probabilities 10^{-18}. With the same computer you get a 2-bit error once every 10^{11} seconds = once every 3,171 years.

There may be other considerations, specifically to do with data communications. It is quite common to establish communications over noisy lines, for example, and then the probability of errors increase dramatically. It often happens that there is a short period when may be multiple-bit errors, and it would be impracticable to use a Hamming code in this situation. Other schemes more appropriate to this problem are used instead.

Exercise:

Find and correct the single bit error (if there is one) in the Hamming code 101101010 using ODD parity.

8.1.3 SECDED Coding

The ability of a Hamming code to correct an error is based on the assumption that only one bit has changed. Changes of two or more bits either are not going to be detected, or will give a false indication of which bits are in error. A common scheme to improve on the situation is called *SECDED Coding* (single-error correct, double-error detect), which makes possible the correction of single-bit errors and the detection, but not correction, of double-bit errors. The original Hamming code is extended by using the unused P0 bit as a parity (odd or even) bit for all the bits in the transmitted symbol. In this way, if one bit is in error, the Hamming checks will fail for a particular set of bits, and also the overall parity will be wrong. We may then assume that there has been a 1-bit error and the Hamming code can correct it. If 2 bits are in error, some Hamming checks will fail but the overall parity will be right, indicating that more that 1 bit was in error but this time we are not able to correct it.

Summarising, using SECDED coding:

- If some Hamming checks fail, and there is also an overall parity error, we assume that there was a 1-bit error and we correct it as before.
- If some Hamming checks fail but there is no overall parity error, we assume that there have been 2 or more errors, and we don't correct them.

Of course, is there have actually been more than 2 errors, the SECDED scheme will give a wrong result. As we have discussed, this is very unlikely.

8.2 Designing Digital Logic Systems

A digital logic system is the translation of a logic expression to a circuit that can implement that logic. Usually it is written first as a Boolean expression or a truth table that provides the Boolean expression. Truth tables may be used when the output is known for all possible combinations of the inputs. From the Truth table the Boolean expression can be written and then simplified. The simplification is important because it reduces the size and complexity of the implemented circuit. The logic circuit is then drawn from the simplified Boolean expression.

Example: The output is true only when the input is less than 2 or greater than 5.

Table 8.2: Truth Table for the example

Row	A	B	C	X	Term
0	0	0	0	1	$\overline{A}\,\overline{B}\,\overline{C}$
1	0	0	1	1	$\overline{A}\,\overline{B}C$
2	0	1	0	0	
3	0	1	1	0	
4	1	0	0	0	
5	1	0	1	0	
6	1	1	0	1	$AB\overline{C}$
7	1	1	1	1	ABC

The resulting Sum of Products is: $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + AB\overline{C} + ABC$

8.3 Binary Adders

8.3.1 Half Adders

To be able to add binary numbers, we need to add individual bits properly, that is, we should be able to calculate their sum with the corresponding carry. The Truth Table for a 1-bit adder — a half-adder — is shown in Table 8.3:

Table 8.3: Truth Table for a Half-Adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 8.1 depicts the logic for a half-adder, implemented with standard gates:

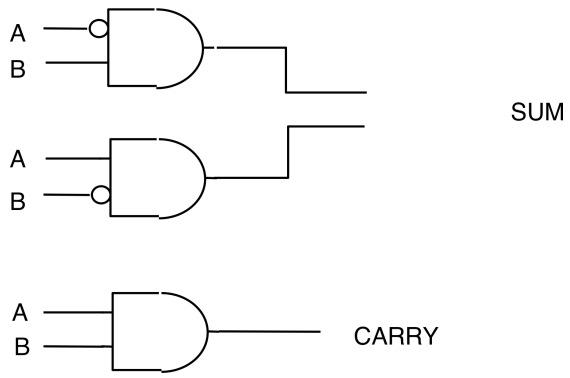

Figure 8.1: Half-adder digital logic

The formulas corresponding to a half-adder are as follows:

Sum = $\overline{\texttt{A}}$.B + A.$\overline{\texttt{B}}$

Carry = A.B

Note that the circuit Figure 8.1 reflects exactly the truth table and the formulas.

8.3.2 Full Adder

In a more complex operation, to add two bits properly we have to make sure not only that we add the two bits, but also that we add the carry properly. The design could be carried out again via a truth table, but a better method is to use two half-adders, since this approach is based on existing components. A full-adder is able to add in the carry from the previous column in a two-digit binary number, as seen in Figure 8.2.

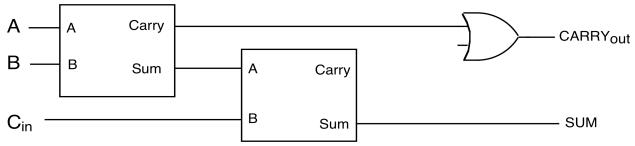

Figure 8.2: Full-adder digital logic

8.3.3 Parallel Adder

A multi-digit adder may be implemented by cascading full-adders, as shown in Figure 8.3. The figure shows 4 full-adders, in which the carry out of each full-adder is fed into the next one. The scheme is initialised with 0 as the first carry in, and the last carry out is the carry out of the whole adder.

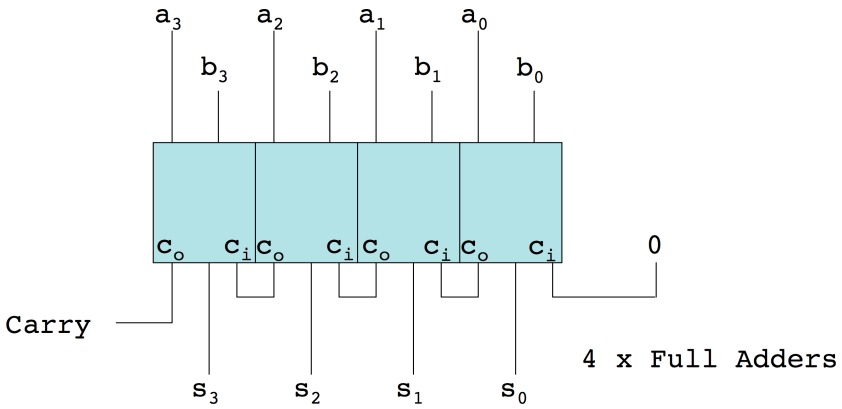

Figure 8.3: Parallel adder digital logic

8.4 Some Digital Components

The design of a function specification such as the ones we have seen must be translated to a combination of digital components. For reasons of cost, these components should be standard, and arranged together in integrated circuits of increasing complexity and size:

- small scale integration, SSI: 1 to 10 components (e.g. logic gates)
- medium scale integration, MSI: 10 to 100 components (e.g. adders, shift registers)
- large scale integration, LSI: 100 to 100,000 component (e.g. arithmetic-logic unit)
- very large scale integration: 100,000+ components (e.g complete microprocessors)

The following section presents some of the most common digital components.

8.4.1 Standard Digital Components

Encoders

An encoder is a digital component that converts 'one-of-many' inputs into a binary coding. Typically there are 2^n inputs and n outputs, for example $2^3 = 8$ inputs and a 3-bit output. An example of encoder use may be to translate the line that is put high by a device seeking to interrupt the processor to the corresponding binary number that is stored as the bit mask of the status vector. The output binary coding represents the number of the input that is active. Figure 8.4 illustrates.

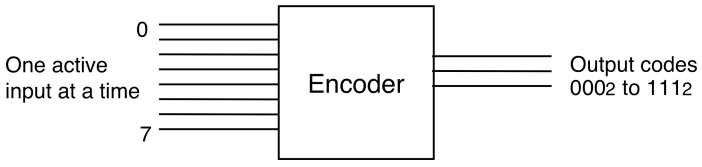

Figure 8.4: A 3-bit encoder

Decoders

A decoder is the opposite to an encoder, where for n inputs there are 2^n outputs, as seen in Figure 8.5.

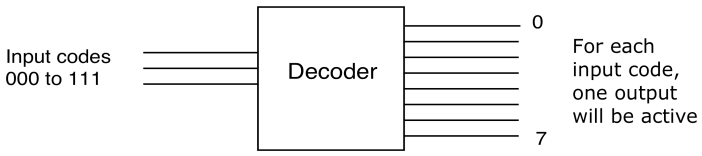

Figure 8.5: A 3-bit decoder

Multiplexor

A multiplexor chooses one input to be connected to a single output. This selects a data source to send to a destination, according to the value of the selector, as shown in Figure 8.6.

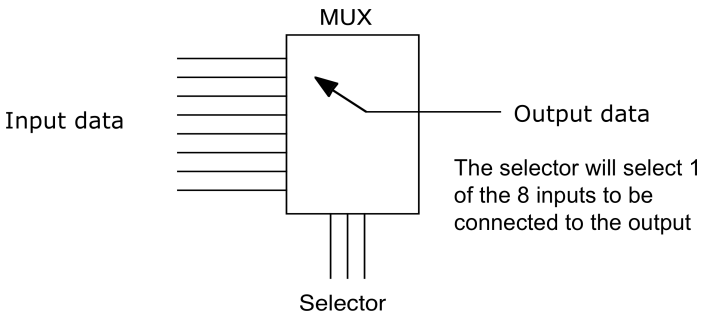

Figure 8.6: A 3-bit multiplexor

Demultiplexor

A demultiplexor directs a single input to one of a number of outputs, according to the value of a selector, as shown in Figure 8.7.

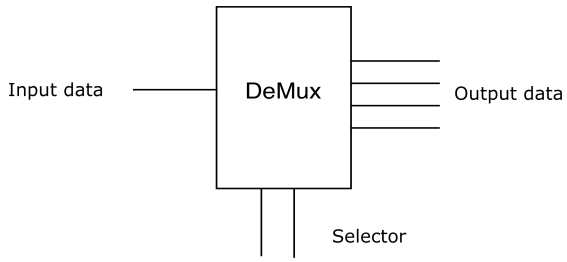

Figure 8.7: A 2-bit demultiplexor

Practical Work 8

8.5 Tutorial exercises

1. Draw the logic diagrams for each of the following Boolean expressions:

   (a) $\overline{P}.\overline{Q}.\overline{W} + R.E.S + \overline{P}.\overline{Q}.\overline{S}$

   (b) $\overline{A}.\overline{B}.\overline{C}.\overline{D} + \overline{A}.B.\overline{C}.D + \overline{A}.\overline{B}.C.D$

   (c) $(A + B).(A + C)$

   (d) $\overline{A}.\overline{B}.(C + D)$

2. Reduce the following Boolean algebra expressions to their simplest *Sum of Products* (expanded—no brackets) form:

   (a) $A.B.C.D + \overline{A}.B.C.D + \overline{B}.\overline{C}.D$

   (b) $(A + B).C + A.\overline{C} + \overline{B}.C$

   (c) $\overline{(B + C).(C + A)}$

   (d) $A.\overline{C} + \overline{A}.B.\overline{C}.\overline{D} + \overline{A}.B.C.D + \overline{A}.B.\overline{C}.D + A.C + \overline{A}.B.C.\overline{D}$

3. Verify the following theorems using a truth table:

   (a) $A + \overline{A}.B = A + B$

   (b) $(A + B).(A + C) = A + B.C$

   (c) $\overline{(A + B)} = \overline{A}.\overline{B}$

4. Convert the string '`PAR`' into 7-bit ASCII with a parity bit where the most significant bit is used to detect errors. Use *odd parity* and express your final answer as a single *hexadecimal* string.

5. Calculate the number of parity bits needed to detect and correct a single-bit error in a string of **8**, **16**, **32** and **64** bits.

6. Determine the single-bit error correction Hamming code using *even parity* for the ASCII string:

   `Oh!`

   - Encode each character individually into its own 11-bit Hamming code.
   - Concatenate the three codes together to form the final binary result.
   - Express that result as a single *hexadecimal* value.

7. An 8-bit ASCII character was encoded as an *even-parity Hamming code* and sent. Unfortunately one of the bits was flipped during transmission. If the code received after transmission was `$34E`, detect and correct the error in the code and determine the original ASCII character.

8.6 Laboratory exercises

With the assistance of your tutor, have a go at developing and testing digital logic diagrams using EasySim or Digital Works. Do diagrams of the items in tutorial exercise 1 and the simplified expressions in question 2.
Use this time to work on your assignment. Consult your lab assistant with any problems you may be having.

Chapter 9

Assembly Language (cont.)

Objectives

After studying this chapter, students should be able to:

- Develop more advanced assembly language programs with further addressing modes, to implement loops
- Describe the structure and function of the 68K stack

9.1 Further Addressing Modes

Up until this point, we have used several addressing modes suitable for the problems we faced:

- Absolute `$3000` or a `label`
- Register direct `dn` or `an`
- Immediate `#`
- Address Register indirect `(an)`

However, there are more addressing modes available for the programmer, which offer further functionality, such as the Address Register indirect with post increment `(an)+`. In this addressing mode the address register is automatically incremented after use, as in:

```
move.b   (a4)+,d3
```

This auto increment saves having to use an extra instruction to modify the address register after access, as we did in previous sample programs.

Code Example 9.1: Read a string from the keyboard

```
;inputs a string of characters from the keyboard into memory
        org     $1000
        move.l  #$7ffe,sp       ;initialise stack pointer
        move.l  #string,a6      ;initialise string pointer
next    move.b  #5,d0           ;get ready to get a char
        trap    #15             ;go get it
        cmp.b   #cr,d1          ;is it a return?
        beq     finish          ;all done if a cr
        move.b  d1,(a6)+        ;put in a6 and increment
        bra     next
finish  move.b  #9,d0           ;prepare to go back
        trap    #15             ;go back

data    org     $1500           ;data starts here
string  ds.b    100             ;space 100 characters
cr      equ     $0d             ;define cr
        end
```

Exercise:

Modify and test this program to include an echo of the input to the display.

This new addressing mode makes it easier to program loops in assembly. Consider the following example that adds three bytes of data:

Code Example 9.2: Add 3 numbers – using a loop

```
;add three bytes of data starting at $1200
start     org         $1000
          move.l      #$7ffe,sp          ;initialise sp
          move.l      #data,a1           ;init address register
          move.b      #3,d2              ;initialise counter = 3
          clr.b       d7                 ;clear result register
loop      add.b       (a1)+,d7           ;get data, add, incr a1
          sub.b       #1,d2              ;decrement counter
          bne         loop               ;do all the adds
          move.b      d7,(a1)            ;save result in memory
          move.b      #9,d0              ;go to OS
          trap        #15

          org         $1200
data      dc.b        4                   ;data space for
          dc.b        12                  ;three data items
          dc.b        7
          ds.b        1                   ;space for result
          end
```

Loops in high-level languages are translated by compilers to assembly loops such as the one above. For instance, the example above is the assembly equivalent of:

```
result = 0;
for (i = 3; i > 0; i--)
{
    result = data[i];
}
```

Exercise:

Draw a memory diagram of this program, including instructions and data. Trace the program first by hand, and then have a look using EASy68K.

9.2 More Addressing Modes

A further addressing mode Address Register Indirect with displacement `d(an)` is convenient when dealing with structures such as the stack. For example, if we write:

```
move.b     4(a4),d0
```

the effective address that goes to the address bus is

```
d + an = 4 + a4
```

This addressing mode is most often used to access elements within consecutive data, such as an array or string.

9.2.1 Sample Program: Reading and Counting a String

Code Example 9.3: Addressing mode example - display 7^{th} character in string

```
1   ;reads and displays the 7th character of a string.
2   ;Test in the lab and observe d0 in the register dump
    ;when the program terminates.
4
            org        $1000
6   start   move.l     #$7ffe,sp      ;init stack
            move.l     #string,a0     ;init pointer
8           move.b     6(a0),d1       ;get char, (7-1)
            move.b     #6,d0          ;send char
10          trap       #15            ;to display
            move.b     #9,d0          ;go back
12          trap       #15

14  ; This part is for the data.
    data    org        $1100
16  string  dc.b       'Hello again!'
            end
```

9.2.2 Program Exercise: Reading and Counting within a String

Code Example 9.4: Count and display number of characters in string

```
1   ;counts and displays the number of characters in a string.
2   ;Fill in the missing branch instruction and the label for the loop
    ;Test in the lab. Fix any bugs or omissions. Limitations?
4   start     org       $1000
              move.l    #$7ffe,sp        ;init stack
6             clr.b     d7               ;zero the counter
              move.l    #str,a5          ;init string pointer
8             move.b    (a5)+,d1         ;get char from string
              cmp.b     #null,d1         ;see if end of string
10            beq       done             ;done, if null
              add.b     #1,d7            ;bump count
12            b..       ...              ;loop till end
    done      add.b     #$30,d7          ;binary to ASCII
14            move.b    #6,d0            ;display count
              trap      #15              ;exit
16  data      org       $1050
    str       dc.b      'count me',null
18  null      equ       0
              end
```

The previous code implements a very common logic, a `while` loop to print the number of characters in a string, as in the following pseudocode:

```
i = 0;
while (str[i] is not NULL)
{
    i++;
}
printline(i);
```

9.2.3 Sample Program Exercise

Write a similar assembly language program to implement the following `while` loop:

```
i = 0;
while (str[i] is not NULL)
{
    print(str[i]);
    i++;
}
printline();
printline(i);
```

9.3 The Stack Revisited: the 68000 Stack

Recall that a generic stack is a LIFO structure, as depicted in Figure 9.1.

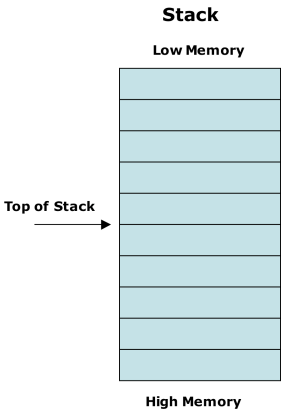

Figure 9.1: A stack

In the 68K assembly, the stack is set aside at the beginning of the code with the instruction:

```
move.l   #$7ffe,sp
```

For this processor, the default stack pointer `sp` is register `a7`. Other address registers may be also used as stack pointers, but `a7` is specifically designed to work properly with the stack. In particular, the `sp` moves in increments of 2, making sure that it is always

pointing to an even address and therefore it is always properly aligned in memory. The 68000 stack can be seen as 16 bits wide, although bytes, words and long words can all be stored.

To store data on the stack we need to implement `push` and `pop` instructions. Consider the assembly program at Code Example 9.3 that includes a sequence of `push` and `pop` instructions:

Code Example 9.5: Count and display number of characters in string

```
1           org             $1000
2           move.l          #$7ffe,sp
            move.b          #$9A,d1         ; set first value to push onto stack
4           move.b          #$35,d3         ; set second value
            move.b          #$F7,d5         ; set third value
6           move.b          d1,-(sp)        ; d0 = 9A
            move.b          d3,-(sp)        ; d3 = 35
8           move.b          d5,-(sp)        ; d5 = F7
            move.b          (sp)+,d2        ; pop to different register so we
10          move.b          (sp)+,d4        ; can see how it comes back
            move.b          (sp)+,d6        ; pop to different register
12  end     move.b          #9,d0
            trap            #15
14          end             $1000
```

This sequence of instructions results in the stack state depicted in Figure 9.2. This is what the stack viewer in EASy68K shows after all of the values have been pushed onto the stack.

As data is popped off, the stack space previously used is available for more data. That part of memory is recycled, and although the data put on the stack remains there, it will be over-written when new data is pushed onto the stack.

When we pop the data off the stack, the order of what we get is reversed, (remember,

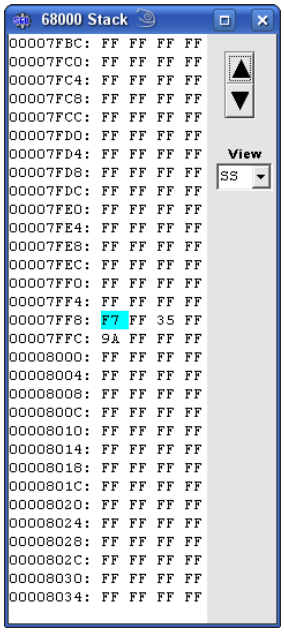

Figure 9.2: Stack display from EASy68K corresponding to sequence

last-in ... first-out). Note in Figure 9.3 that the contents of register d2 is the same as d5, and that d6 and d1 are the same. This is because we pushed d1 first, then d3, then d5; and when we popped the data, we popped d2 first, thus getting the last value pushed onto the stack.

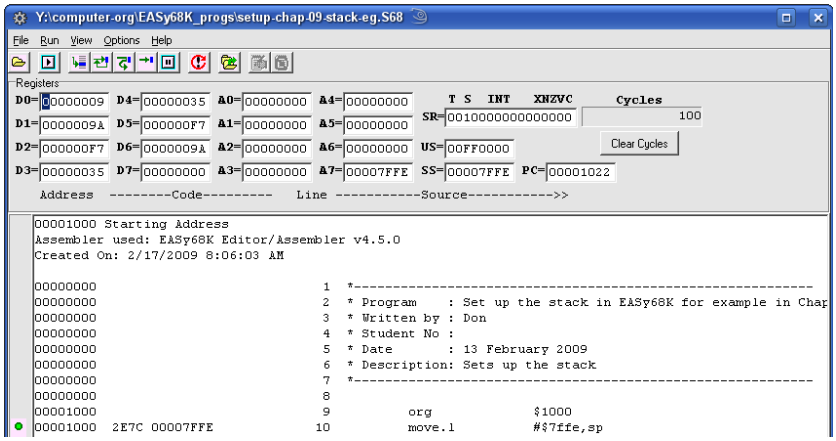

Figure 9.3: Register contents after popping data from the stack

As a consequence of the required memory alignment, the stack pointer is always pointing to an even memory address. Thus, a byte only uses a byte size location and leaves a byte size location empty. Although this is a loss of space, it is really negligible with large modern memories. Looking again at Figure 9.2 you can see that this is what happened.

Practical Work 9

9.4 Tutorial exercises

1. Design a minimised SOP logic system for a system with four inputs representing a 4-bit binary number. The system output is TRUE when the binary input is:

   (a) Less than 3_{10}

   (b) Equal to 12_{10}

   (c) Greater than 13_{10}

2. Alogic system has four inputs (A, B, C, and D) and a single output (X). X is true when the inputs total 0, 2, 3, 10, 11, 14, or 15.

   - Use a truth table to obtain the sum-of-products expression.
   - Minimise the expression using Boolean algebra.

3. A keystroke counting program

   (a) Write a program to take a string from the keyboard into memory, keeping a count of the keys. Your program should terminate when the user presses the Enter key. Assume first that the string is shorter than 9 characters. Print the count at the end.

   (b) Modify the program above so if the key count gets to 15 the program terminates. Print the count at the end.

9.5 Laboratory exercises

Create and test in the lab the programming exercises discussed in the tutorial.

Chapter 10

Digital Logic(cont.)
Assembly Language (cont.)

Objectives

After studying this chapter, students should be able to:

- Describe standard memory elements such as Flip-Flops
- Explain how standard elements may be used to build registers, counters and dividers
- Explain the notion of subroutines and the related instructions `bsr` and `rts`
- Describe the characteristics of the 68K stack and its to pass parameters
- Explain how subroutines may retrieve parameters from the stack

10.1 Memory Elements

As discussed before, there are two kinds of memory elements in digital logic systems:

- Flip-Flops (used in static memory)
- Capacitors (used in dynamic memory)

10.1.1 The SR Flip-Flop

The most common flip-flop is the SR flip-flop. The truth table of the SR FF is given below:

Table 10.1: SR Flip-Flop truth table

Set	*Reset*	Q
0	0	No change
0	1	0
1	0	1
1	1	Undefined

10.1.2 Clocked Flip-Flops

Due to their large number and speed, there is a need to coordinate computer operations. All computers use a *hardware clock* to properly sequence operations, making sure that circuits are in a stable, correct state before using their outputs. A clock is a hardware ocillator that (ideally) produces a very regular, square wave like so:

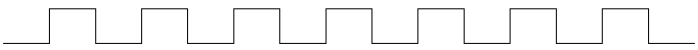

Figure 10.1: A clock wave

If we synchronise all activity associated with a state change with clock cycles, we are able to ensure that there is enough time for the circuits to change state and be stable before the next step.

The time between two rising edges is called the *clock cycle time* — also known as a *clock tick* — measured in Hertz (1Hz = 1 cycle per second). Typically a modern processor has a clock cycle of 1, 2, 3 Giga Hz (billions of cycles per second), that is, the clock ticks 1, 2, 3 billions of times per second. In general, the faster the clock the faster the computer, but the relationship is not completely direct, since an instruction may take many clock cycles.

With FFs, the time that takes for the logic to flow through makes them difficult to use in circuits. Thus, flip-flops may be coordinated by the use of a clock to control their downstream logic flow. *Clocked flip-flops* use extra logic to make them more useable in logic systems. Two common types of clocked FFs are the D and JK FFs, shown in Figure 10.2 in the up (positive) clock cycle.

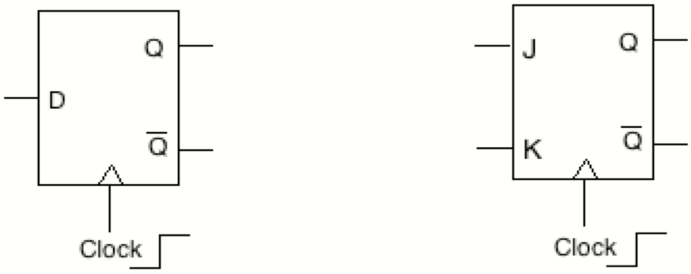

Figure 10.2: The D and JK clocked FFs

Clocked FFs gate their inputs to the output only when a clock signal occurs. In this way, the clock synchronises data transfers within a computer.

10.1.3 The D Flip-Flop

The operation of a D FF is quite simple: the logic level on the D input is transferred to the output Q when the clock tick occurs. This is often used when there is a need to transfer binary information in synch with the clock. For example, in computer registers, counters and shift registers to store multi-digit binary data. Table 10.2 shows the truth table for a D FF.

Table 10.2: The D Flip-Flop truth table

D	Q_{n+1}
0	0
1	1

10.1.4 The JK Flip-Flop

JK FFs are more complex, but they are also more flexible in their use. Table 10.3 shows the truth table of the JK FF.

Table 10.3: The JK Flip-Flop truth table

J	K	Q_{n+1}	Result
0	0	Q_n	no change
0	1	0	reset
1	0	1	set
1	1	$\overline{Q_n}$	toggle

10.2 Registers

A register, such as a CPU register, is a group of FFs arranged to receive and store multi-digit binary data. The input data is stored in the register on the clock. The stored data can be read at the outputs at the next tick of the clock. In this way, the clock synchronises transfers between registers and between registers and memory. Static memory works like this, using D FFs. See Figure 10.3.

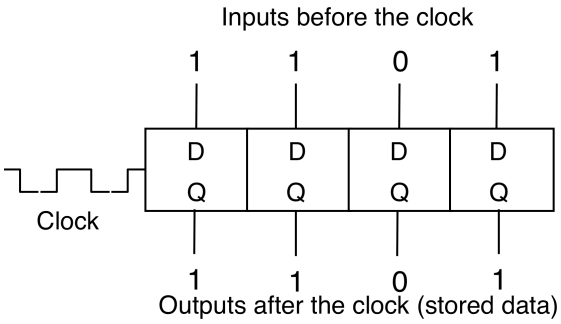

Figure 10.3: A register made out of D FFs

Shift registers are organised so that on each clock input, the data stored in the register moves one position to the right. This operation is used in computers to translate parallel data into serial data, for example to transfer data in serial format over long distances, such as the Internet. This is known as *serial data transfer*. The diagram

corresponding to a Shift Register is shown in Figure 10.4.

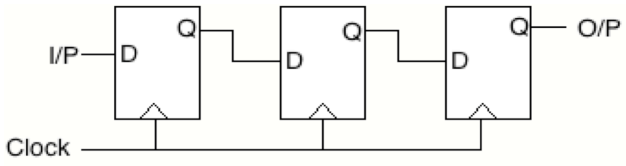

Figure 10.4: A shift register

If the Flip-Flops are clocked on the positive edge of the clock, every time the clock wave goes up the FFs change, moving the bits from one FF to the next. The diagram in Figure 10.5 shows the effect on the register of a HIGH input signal for a brief period of time (i.e. the input of a 1).

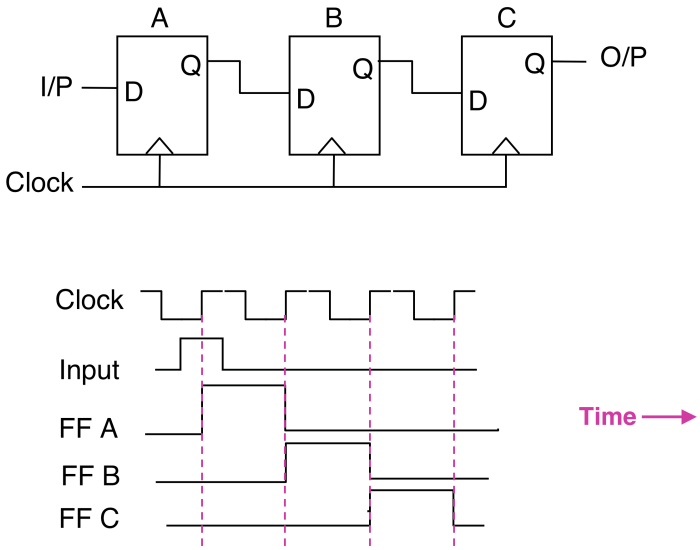

Figure 10.5: A time series for a shift register

The waves in the figure refer to the state of each individual FF as the signal goes through the register.

Student Exercise:

Draw the logic diagram and the waveforms for a 4-bit shift register. The initial outputs are 0101 and the input is 1 for a short period during the first clock edge.

Laboratory Exercise:

Build this Shift Register with EasySim and verify the operation. Use a switch for the clock and lamps for each of the FF, Q outputs.

10.3 Counters and Dividers

Consider the device in Figure 10.6. It consists of 3 JK Flip-Flops, where the first FF clock input is connected to the clock, but each following FF clock input is connected to the $\overline{Q}$ output of the previous FF. All FFs have both J and K inputs 1 (HIGH), so each time an input is received in the clock, the FF toggles. Assume that the FFs are clocked on the positive edge of the clock, and that all Q outputs are initially zero.

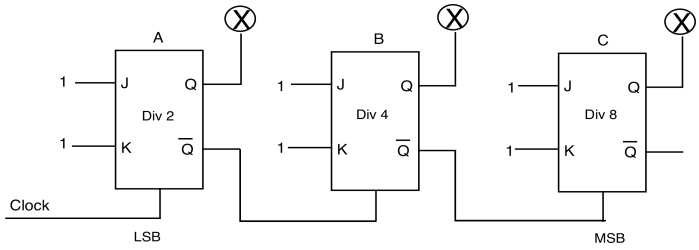

Figure 10.6: A signal divider

This results in the first FF toggling each time the clock goes high. The second FF goes high on the positive edge of the output of the first FF, that is, when the $\overline{Q}$ output of the first goes from 0 to 1. Hence, when the first clock positive edge produces the change of the first FF Q output from 0 to 1, the first FF $\overline{Q}$ goes from 1 to 0 and so the second FF does not toggle. However, when the second positive edge of the clock is input into the first FF, the FF toggles again, and as a consequence $\overline{Q}$ goes from 0 to 1, and this time the second FF toggles Q from 0 to 1. As a consequence, the second FF

only toggles each second time the first one does, and then the second FF keeps the 0 and 1 states double the time the first one does. Figure 10.7 shows the sequence of the first 8 states of the device.

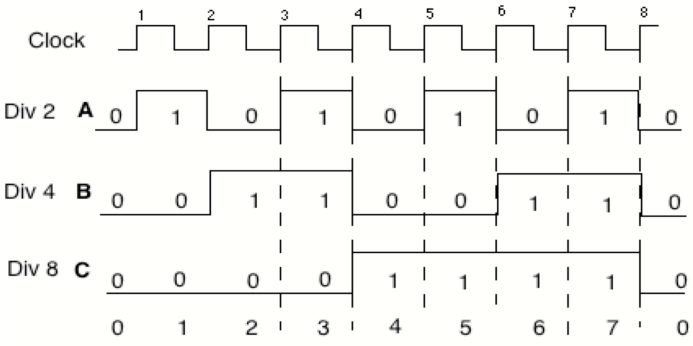

Figure 10.7: A divided wave

It is easy to see that the original clock wave frequency has been divided by 2 in the first FF, it has been divided by 4 in the second FF and it has been divided by 8 in the third FF. Consequently, this device is usually called a *divider*. By looking at the output of a single FF we obtain the original frequency divided by 2, 4, 8, etc. Naturally, if the FFs were clocked on the negative edge of the clock (i.e. from 1 to 0), we would obtain the same behaviour by using the Q output of each FF rather than $\overline{Q}$.

There is also another use for this type of devices, if we observe the output of all the FFs rather than each individually. We can see that the output of the 3 FFs form a binary increasing sequence, if we consider the first FF as the LSB. Reading the number that way, we get the sequence:

```
000
001
010
011
100
101
```

```
110
111
```

That is, we have an 'up counting' binary sequence. If we look at the $\overline{Q}$ outputs, we obtain a 'down counting' sequence instead.

10.4 Subroutines

Subroutines are implemented so that only a single copy of a frequently used piece of code is required within a program. Subroutines save programmer time by avoiding having to rewrite the same code all the time. They also save memory space because there is only one copy; a routine that uses 100 bytes of memory and is used 1000 times would take 100,000 bytes of memory instead of only 100 bytes.

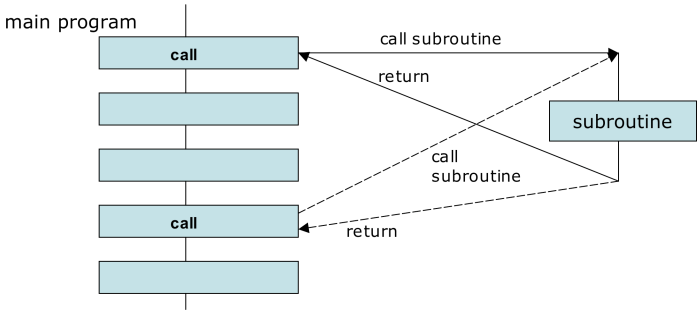

Figure 10.8: Main program calling a subroutine twice

When the main program finds a subroutine call, it suspends execution and takes appropriate action to call the subroutine. Hence, while the subroutine is executing the main program is suspended. At the end of its execution, the subroutine returns control to the main program, which then resumes execution. Figure 10.8 illustrates the main program calling a subroutine twice.
In assembly, programs call subroutines using `bsr` (branch to subroutine), and use `rts` (return from subroutine) to return to the caller.

- `bsr` does two things:

  - pushes the *return address* onto the stack; the return address is the address of the instruction *after* the `bsr` instruction;
  - puts the starting address of the subroutine into the PC, so the next instruction to be executed is the first instruction of the subroutine.

- The `rts` instruction pops the return address from the stack into the PC; this restores execution to its original position.

The following code illustrates the use of subroutines to store keyboard input into a string.

Code Example 10.1: Keyboard input using subroutine

```
1   ;inputs keys into a string using subroutines.Test in lab.
2   code    org       $1000         ;code space
            move.l    #$7ffe,sp     ;stack at the top memory
4           move.l    #keys,a2      ;string pointer
    next    bsr       input         ;get a key
6           cmp.b     #cr,d0        ;is this the end of input
            beq       exit          ;exit on cr
8           bsr       output        ;print the key
            move.b    d0,(a2)+      ;put key into string
10          bra       next          ;loop till cr
    exit    move.b    #9,d0         ;exit
12          trap      #15

14  input   move.b    #5,d0          ;subroutine to read a key
            trap      #15
16          rts                      ;return to caller
    output  move.b    #6,d0          ;subroutine to display key
18          trap      #15
            rts                      ;return to caller
20  keys    ds.b      100            ;reserve 100 bytes for string
```

```
    cr      equ     $0d
22          end     $1000
```

However, the code above has a problem, because if more than one code segment — such as main or a subroutine — is modifying the contents of the `d0` and `d1` registers, the contents of these registers may depend on the order in which the segments are executed. This characteristic is described by saying that the code is *not reentrant*.

```
; the problem with the previous code is that the contents
; of d0 and d1 are manipulated without any care, so the
; code is not re-entrant. Suggested modification:
......
.........
input   ????                    ; push d0 on the stack
        ????                    ; push d1 on the stack
        move.b      #5,d0       ; subroutine to read a key
        trap        #15
        ????                    ; restore d?
        ????                    ; restore d?
        rts                     ; return to caller

output  ????                    ; push d0 on the stack
        ????                    ; push d1 on the stack
        move.b      #6,d0       ; subroutine to display key
        trap        #15
        ????                    ; restore d?
        ????                    ; restore d?
        rts                     ; return to caller
......
.........
```

Consider the instructions required in the above modified version, and particularly consider the *order* of the "pop" instructions.

Student Exercises

For the following problems, draw flow charts, write the corresponding programs and test them. Use subroutines where possible.

1. Modify the program above to make it re-entrant as shown in the suggested modification.

2. Add 5 words of data in memory and save result.

3. Print a message on the display and then input the user response. Display the response and store it in memory. Examine the string in memory with `MD`.

10.5 Parameter Passing

The primary reason for the existence of subroutines is the need to repeat the same processing steps on a number of different data values. Consider the small programs that we have been examining which add a series of numbers. What would be the easiest way to re-write one of these "sum" programs so that, instead of a simple sum of three mumbers, it produced the sum of the *square* of each of the three numbers?
We could write the program so that it:

```
move.b number,dx
mul.b      dx,dx
add.w      dx,dy
move.b     number2,dx
mul.b      dx,dx
add.w      dx,dy
move.b     number3,dx
mul.b      dx,dx
add.w      dx,dy
```

But this code is starting to have big blocks of instructions that are essentially identical, except for the data that they are acting on. What is needed is a way to write the code in a "general" format and then, at "run-time" pass the correct data to the code block for processing.
Fot this reason a subroutine often needs a way of getting data and returning results to a calling program. To do this, it may be necessary to pass *parameters* or *arguments* to the subroutine before it executes. This technique is known as *parameter passing*. Parameters may be passed by three different methods:

1. In registers.
2. In memory
3. On the stack

Passing parameters on the stack is preferred, since it has virtually no space limitations and prevents errors due to data in registers or in memory being altered by other parts of

the code, and thus is a 'fully re-entrant' method. To pass them on the stack, parameters are pushed on the stack and then the routine is called.

```
move.w    #6,-(sp)      ;push parameter 1
move.w    data,-(sp)    ;push parameter 2
bsr       xyz           ;subroutine call
```

Once the parameters have been pushed on the stack, the subroutine accesses them, but carefully since the `bsr` instruction has pushed the program counter PC on the stack before replacing it with the subroutine starting address. The state of stack on entry to the subroutine — i.e. after the call — is shown in Figure 10.9.

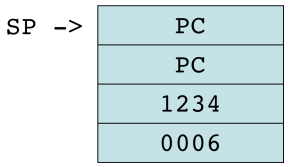

Figure 10.9: Parameter allocation in the 68K stack

Very important things to note in this code:

- Both parameters are integer numbers, but they have been passed on the stack in different ways:
  - parameter 1 is an integer, immediately passed on the stack in the instruction itself, so the value of the parameter is pushed on the stack
  - parameter 2 is also an integer, but it has been passed via its address in memory, that is, the location of the data is copied on the stack
- The PC return address is 2 words, and each of the two parameters (in this case) are 1 word. The subroutine needs to know the order in which the parameters have been pushed and their type, so it can retrieve them properly.

When called, the subroutine can use the addressing mode `d(An)` to access each word on the stack. The distances of the words from the top of the stack pointed by `sp` are 0, 2, 4 and 6:

```
move.w      4(sp),d5          ;get parameter 2
move.w      6(sp),d6          ;get parameter 1
```

Please note that the value of the PC has been skipped when retrieving the parameters.

The following code exemplifies passing and retrieving parameters on the stack. The status of the stack is depicted in Figure 10.10.
Look at the information available in the main window from the EASy68K simulator(Figure 10.11). Look for the address of the next instruction following the `bsr` instruction. See if you can find this address in the stack in Figure 10.10.

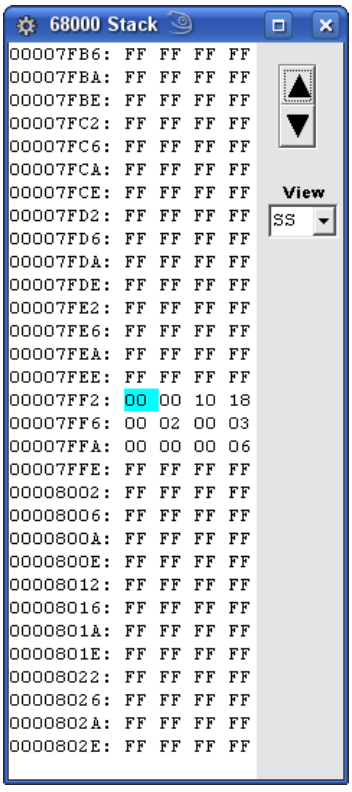

Figure 10.10: Passing parameters on the 68K stack

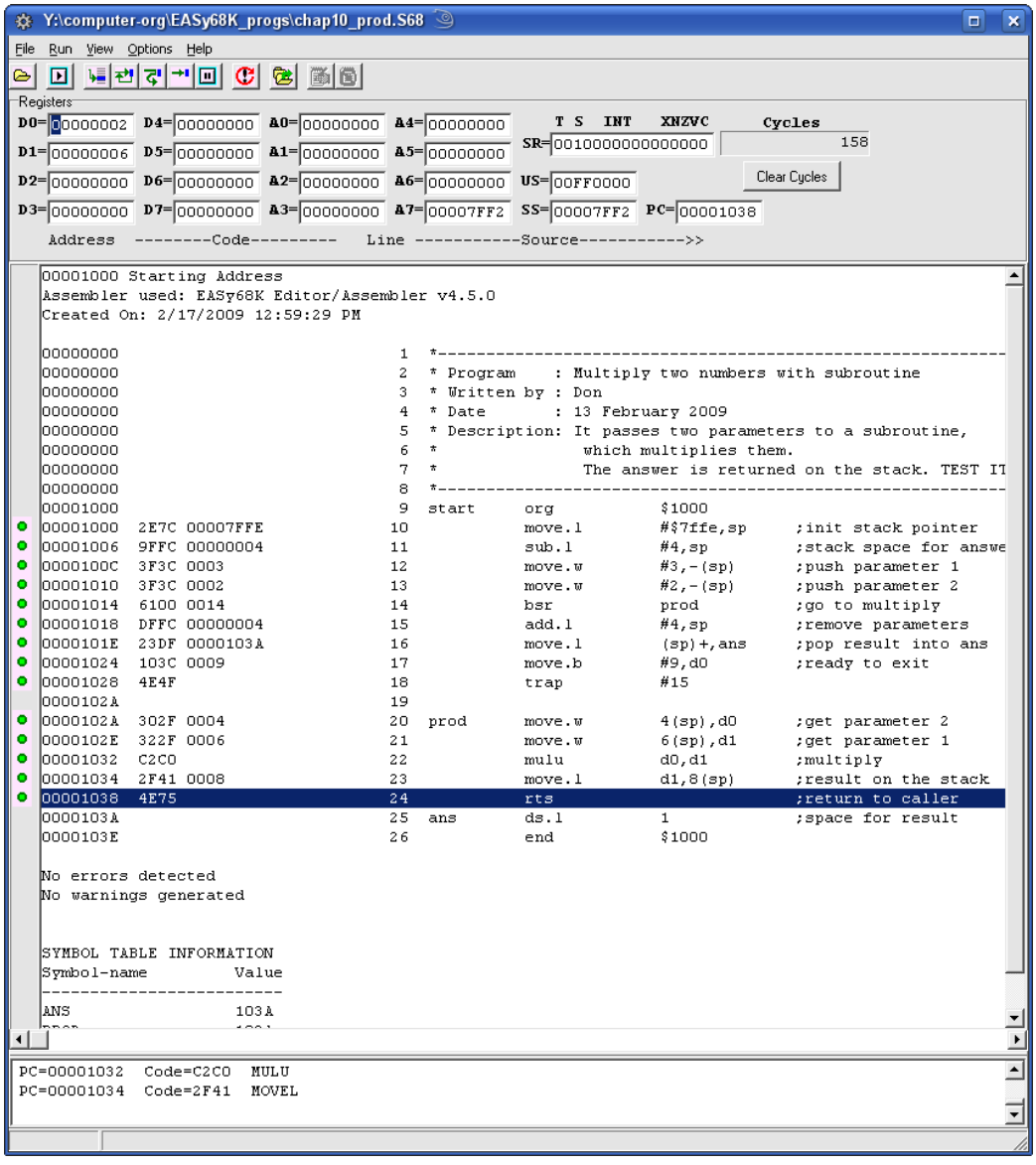

Figure 10.11: Passing parameters on the 68K stack - main window

Code Example 10.2: Parameter passing via the stack

```
1   ; It passes two parameters to a subroutine,
2   ; which multiplies them.
    ; The answer is returned on the stack. TEST IT.
4   start     org          $1000
              move.l       #$7ffe,sp     ;init stack pointer
6             sub.l        #4,sp         ;stack space for answer
              move.w       #3,-(sp)      ;push parameter 1
8             move.w       #2,-(sp)      ;push parameter 2
              bsr          prod          ;go to multiply
10            add.l        #4,sp         ;remove parameters
              move.l       (sp)+,ans     ;pop result into ans
12            move.b       #9,d0         ;ready to exit
              trap         #15
14
    prod      move.w       4(sp),d0      ;get parameter 2
16            move.w       6(sp),d1      ;get parameter 1
              mulu         d0,d1         ;multiply
18            move.l       d1,8(sp)      ;result on the stack
              rts                        ;return to caller
20  ans       ds.l         1             ;space for result
              end
```

Practical Work 10

10.6 Tutorial exercises

1. A programming exercise

   (a) Write a single assembly language program that:

   - Will allow the user to enter a string of up to *100* characters and store it in memory.
   - Your program must echo each character to the screen and store it in memory as it is typed.
   - The user will press Enter to terminate the input (`INCH` returns `cr` in this case).
   - You must terminate the string in memory with a `null` character.
   - Once the user has terminated the string, the program must loop through the string stored in memory and print all the characters.

   (b) Modify the program above to check the input and make sure that only digits and letters are accepted and stored. All other characters should be ignored.

   (c) Modify the program above to only print the letter characters.

   (d) Modify the program above to print the uppercase version of the all the characters.

2. Draw the logic symbol and truth table of:

   - a *D* flip-flop
   - a *JK* flip-flop

3. Draw the logic diagram and waveforms of an *asynchronous counter* using 4 JK flip-flops.

   - How many numbers can this counter represent?
   - How can you use this counter to count backwards?

4. Draw the logic diagram and waveforms of a 4-bit *shift register* with initial values $(1, 0, 0, 1)$. The input should begin at *0* for the first clock edge and then remain at *1* for the remaining clock edges. Draw the waveforms for 8 clock edges.

10.7 Laboratory exercises

Use this time to work on Assignment 3. Consult your lab assistant with any problem that you may have.

Chapter 11

Assembly Language (cont.)

Objectives

After completing this chapter, students should be able to:

- Develop more complex assembly language programs with advanced addressing modes, subroutines and parameter passing

11.1 An Assembly Language Project

Assume that a project requires to write a single assembly language program that will meet the following requirements.

Requirement One

Allow the user to enter a string of up to *100* characters. You must echo each character to the screen and store it in memory as it is typed. At the end of the input the user will press Enter, (`INCH` returns `cr` in this case). You must terminate the string in memory with a `null` character. (**Note:** while it is true that EASy68K provides a trap function that will automatically read and store a string, along with echoing the characters as they are typed, we will not be using these functions in the simple examples presented in these notes. ***and*** *you* will not be using these advanced functions for either assignment one or assignment two.)

Requirement Two

Once the user has terminated the string, your code must loop through the string stored in memory and count the number of "words" that appear in the string. For the purposes of this count, a word is defined as any series of alphanumeric characters (`a`–`z`, `A`–`Z` and `0`–`9`) which is separated by blanks from the adjoining words.

E.g. '`Fred can, run!?!`' contains **3** words ('`Fred`', '`can`' and '`run`').

E.g. '`What's that`' also contains **3** words ('`What`', '`s`' and '`that`').

E.g. '`$AU 123.466`' also contains **3** words ('`AU`', '`123`' and '`466`').

Requirement Three

You are also required to keep the position of the first character of the longest word in the string. In the case that multiple words are of the same length, you must keep the position of the first occurrence of a word of that length.

E.g. '`This is a short word`' must maintain a pointer to '`short`'.

E.g. '`A long sentence contains long words`' must maintain the first position of '`sentence`'.

Requirement Four

Once the loop has been completed you must display both the number of words (in multi-digit *hexadecimal*) and the longest word on the screen with an appropriate message, e.g.–

```
There were 3 words in the string
The longest word was:  fred
```

11.2 How Do We Develop This Project?

One of the main issues in programming is that as the complexity and size of the programs increases it is necessary to tackle the problem in a stepwise fashion. It is simply not possible to write all the required code in one go. This approach requires careful planning, and making sure that each step is implemented and fully tested before making further progress. To illustrate the approach, we are going to suggest a development strategy for the project described above, as follows:

1. Allow the user to enter a string in memory, and terminate the program when this is done. Make sure that the string is correctly stored in memory, including the null character at the end. This ensures that your input is OK. (**Note:** with EASy68K you can easily watch the destination memory to make sure that your program is getting this right.)

2. Loop through the string and display it by displaying each character at a time. Exit the program at the end of the displaying. This makes sure that your loop through the string works properly.

3. Modify the code to print the first word. This makes sure that you are able to print a word properly.

4. Loop through the string and display a word at a time, one word per line. This ensures that your program properly recognises each word, their beginning and end.

5. Write a subroutine to display a number in its hexadecimal form. Test in a separate program.

6. Include the subroutine into the code, and count and display the number of words.

7. Modify the code to print the length of each word together with the word one per line. This makes sure that you are getting the length of the words right.

8. Modify the code to keep and print the length of the longest word.

9. Modify the code to print the longest word by keeping a register pointing to the longest word.

11.3 Coding Style

You should use defined constants as much as possible, and subroutines when the behaviour of a code section can be separated out. Another, less obvious benefit of using constants is that, if the code is moved to a system that uses different numbers for functions or different values, it is only necessary to make one change to convert the program. The following are constants that may be used for this project:

Code Example 11.1: Example of constant declaration

```
1  ;*****************************************************************
2  ;* CONSTANTS
   ;*****************************************************************
4
   LINE_BREAK    dc.b   CR,LF,NULL
6
   PROMPT        dc.b   'Please enter up to 100 characters (Enter when '
8                dc.b   'finished):',NULL
   THERE_WERE    dc.b   'There were ',NULL
10 WORDS         dc.b   ' word/s in the string',NULL
   LONGEST_WAS   dc.b   'The longest word was: ',NULL
12
   ; ** Assembler constants ************************************
14 ; The values defined below may be used throughout the code to make
   ; that code clearer.  Compare for instance the following two lines:
16 ;
   ;       add.l  #4,sp
18 ;       add.l  #LONGWORD_LEN,sp
   ;
20 ; In the first example you may wonder why 4 is being added to the
   ; stack pointer (and not 3, 5 or some other).  The second example
22 ; suggests that the aim is to remove a longword from the
   ; top of the stack (increasing the stack pointer so this value sits
```

```
; above the usable stack area). Now compare these two snippets:
;
;                   move.b  #5,d0           move.b  #INCH,d0
;                   trap    #15             trap    #OSCALL
;
; Both achieve the same thing, but the right-hand example is more
; obvious because it uses constants instead of 'magic
; numbers' (numbers whose purpose is unknown) .
;
; Always make good use of constants!
;************************************************************

LONGWORD_LEN  equ  4      ; the length of a longword in bytes
WORD_LEN      equ  2      ; the length of a word in bytes

NULL          equ  $00    ; used to indicate the end of a string
CR            equ  $0D    ; ASCII code for carriage return
LF            equ  $0A    ; ASCII code for line feed

QUIT          equ  9      ; to indicate that control of the
                          ; computer should return to the OS
INCH          equ  5      ; OS to read from the keyboard
OUTCH         equ  6      ; OS to print to the display
OSCALL        equ  15     ; number 'trap' needs to perform OS calls
```

In addition, the following routine may be used to read a character from the keyboard:

Code Example 11.2: Keyboard input and display output routines

```
;* get_char ************************************************
;  Waits for a character from the keyboard and then places its
;  ASCII value into the lowest byte of d1.
;  Uses d0 non-destructively
;***********************************************************

get_char    move.l  d0,-(sp)    ; save d0's original state
            move.b  #INCH,d0    ; tell the OS to read a character
            trap    #OSCALL     ; call the OS
            move.l  (sp)+,d0    ; restore d1's original state
            rts                 ; return to caller
```

And this one may be used to print a character on screen:

```
;* print_char  *********************************************
; Prints the character to the screen whose ASCII character is
; found in the lowest byte of d1.  Uses d7 non-destructively
;***********************************************************

print_char  move.l  d0,-(sp)    ; save d0's original state
            move.b  #OUTCH,d0   ; tell the OS to read a character
            trap    #OSCALL     ; call the OS
            move.l  (sp)+,d0    ; restore d0's original state
            rts                 ; return to caller
```

The following uses the previous ones to read the string into memory:

Code Example 11.3: Keyboard string input routine

```
\begin{verbatim}
;* get_string ************************************************
; Reads a sequence of characters, storing them in the memory location
; specified on the stack, until the Enter key is pressed.
; The ASCII code associated with the Enter key (CR) is not stored.
;
; This input subroutine is primitive.  For instance, every
; key except for Enter is saved to memory (including backspaces),
; the user may enter more than 100 characters; this
; means that the user could corrupt surrounding memory.
;************************************************************

get_string    move.l  a2,-(sp)      ; save the states of registers
              move.w  d0,-(sp)      ; that will be used

              move.l  10(sp),a2
gs_loop       bsr     get_char
              cmp.b   #CR,d0
              beq     gs_done
              bsr     print_char
              move.b  d0,(a2)+
              bra     gs_loop

gs_done       bsr     line_break    ; move to the next line
              move.b  #NULL,(a2)    ; terminate the string with a NULL

              move.w  (sp)+,d0      ; restore register states
              move.l  (sp)+,a2
              rts                   ; return to caller
```

The following may be used to print a string:

Code Example 11.4: String output routine

```
1   ; ** print_string ***********************************************
2   ; *  Prints a sequence of characters from memory until a NULL
    ; *  character is encountered.
4   ; *  Uses registers:  a2 - holds the address of the next character
    ; *                   to be examined
6   ; *              d0 - any characters to be printed are copied here
    ; *                   for passing to the OUTCH subroutine
8   ; *
    ; *  Modifies registers:  None.  All registers are restored when
10  ; *                       the subroutine completes.
    ; ****************************************************************
12
    print_string   move.l   a2,-(sp)   ; save the current contents of a2
14                 move.w   d0,-(sp)   ; save part of d0
                   move.l   10(sp),a2  ; read the address of string to be
16                                     ; printed into a2

18  ps_loop        move.b   (a2)+,d0   ; read byte at current address,
                                       ; increment address register
20                 cmp.b    #NULL,d0   ; is that byte a NULL?
                   beq      ps_done    ; finish up if so
22                 bsr      print_char ; print the character to screen
                   bra      ps_loop    ; repeat as needed
24
    ps_done        move.w   (sp)+,d0   ; restore d0's original state
26                 move.l   (sp)+,a2   ; restore a2's original state
                   rts                 ; return to caller
28
```

Finally, the following routine counts the words:

Code Example 11.5: Routine to count "words"

```
; ** count_words ***************************************************
; *  Scans across a NULL-terminated string in memory, keeping
; *  count of the words found.  A word is defined as a consecutive
; *  sequence of word characters separated by at least
; *  one non-word character.
; *  The stack should look like this before calling the subroutine:
; *                  TOP
; *              -----------
; *              | xx | xx |  address of starting character of
; *              | xx | xx |  string
; *              -----------
; *              | xx | xx |  location where result will be stored
; *              -----------
; ******************************************************************

count_words     move.l  a2,-(sp)
                move.w  d0,-(sp)        ; save the original registers
                move.w  d1,-(sp)        ; that will be used
                move.w  d2,-(sp)

                move.l  14(sp),a2       ; a2: current location in string
                clr.w   d2              ; d2: running word count

cw_findstart    move.b  (a2)+,d0        ; get the next character
                cmp.b   #NULL,d0        ; break out if end of string
                beq     cw_done         ; has been reached
                bsr     is_word_char
                cmp.b   #0,d1
                beq     cw_findstart    ; until a word character is found
```

```
30
                add.w   #1,d2           ; increment the word count
32
    cw_findend  move.b  (a2)+,d0        ; get the next character
34              cmp.b   #NULL,d0        ; break out if end of string
                beq     cw_done         ; has been reached
36              bsr     is_word_char
                cmp.b   #0,d1           ; until a non-word character is
38              bne     cw_findend      ; found

40              bra     cw_findstart

42  cw_done     move.w  d2,18(sp)       ; save the count to the stack

44              move.w  d2,(sp)+
                move.w  d1,(sp)+        ; restore register states
46              move.w  d0,(sp)+
                move.l  a2,(sp)+
48              rts                     ; return to caller
```

Practical Work 11

11.4 Tutorial exercises

1. The following instruction is equivalent to the combined effect of two statements seen in the past. What two statements are these?

```
move.b      (a3)+,d2
```

2. What will be the output of the following program?

Code Example 11.6: A test program– prac 11.1

```
1           org         $1000
2           move.l      #$7ffe,sp
            move.l      #ints,a0
4           clr.b       d3
    loop    move.b      (a0)+,d4
6           cmp.b       #null,d4
            beq         end
8           add.b       d4,d3
            bra         loop
10  end     move.b      #outch,d0
            trap        #15
12          move.b      #exit,d0
            trap        #15
14  data    org         $1200
    ints    dc.b        '7654321',null
16  null    equ         00
    outch   equ         6
18  exit    equ         9
            end         $1000
```

3. What would the output be after replacing the fifth line with the following:

```
loop    move.b      +(a0),d4
```

4. What is the value stored in a3 after the following program terminates?

Code Example 11.7: A test program– prac 11.2

```
1           org         $1000
2           move.l      #2000,a3
            move.b      #6,(a3)+
4           move.b      #7,(a3)+
            move.w      d2,(a3)+
6           move.l      d4,(a3)+
    end     move.b      #9,d0
8           trap        #15
            end
```

5. What is the value of the stack pointer (a7) after the completion of the following program?

Code Example 11.8: A test program– prac 11.3

```
1           org         $1000
2           move.l      #$7ffe,sp
            move.w      #179,d2
4           move.l      #2148,d4
            move.b      #6,-(sp)
6           move.b      #7,-(sp)
            move.w      d2,-(sp)
8           move.l      d4,-(sp)
    end     move.b      #9,d0
10          trap        #15
            end
```

6. Show the data stored in memory at the completion of the program assuming the following residual memory shown in Figure 11.1:

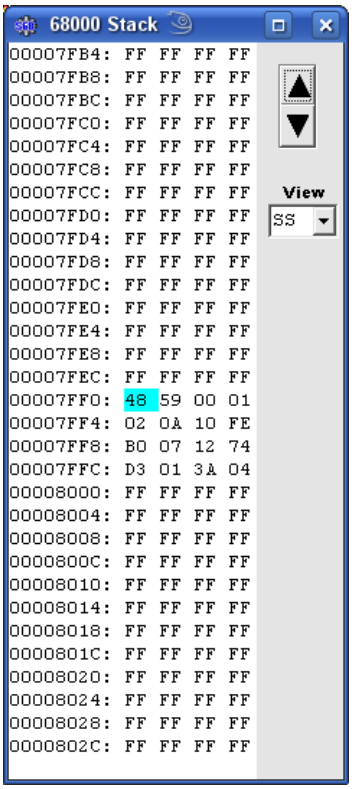

Figure 11.1: Stack listing at program termination

7. Assuming that the above program had executed, what would be the contents of `d3` after the following operations?

(a) `move.b 2(sp),d3`

(b) `move.w 6(sp),d3`

(c) `move.l 4(sp),d3`

(d) `move.b 3(sp),d3`

8. In reality you should never use *address register indirect* on the stack pointer with an odd offset. Why?

11.5 Laboratory exercises

Use this time to work on Assignment 3. Consult your lab assistant with any difficulties you may be experiencing.

Chapter 12

Assembly Language (cont.)

Objectives

After completing this chapter, students should be able to:

- Develop more advanced assembly language programs with further addressing modes, subroutines and parameter passing
- Explain the concept of activation records and frame pointer
- Describe the mechanism of nested subroutine calls
- Explain the consequences of the concepts above for variable scope

12.1 Parameter Passing in High-Level Languages

In a previous chapter we saw parameters being passed on the stack either by pushing the value itself on the stack, or by pushing the address of the parameter on the stack. Let us consider now two `int` variables `x` and `y` that have been declared by a programmer in a high-level language

```
int x, y;
```

and that they are now passed as parameters to a function or method. Since the variables have been declared, the compiler allocates room in memory for the `int`s, and allows the program to refer to them by their names `x` and `y` instead of their locations (which are unknown to the programmer). If the programmer wants to make these variables

available as parameters to a function, there are two different ways of doing so: passing the parameters *by value*, or *by address* (also called *by reference*). As discussed, passing a variable as a parameter by value pushes the value of the variable on to the stack, and then the copy of the value is manipulated by the subroutine. In contrast, passing a variable as a parameter by reference pushes the address of the variable on to the stack.

The critical difference between passing by value and passing by reference is that in passing by reference, it is possible for the subroutine (function, method) to modify the value of the *original* variable. In passing by value, the original value remains unchanged by the subroutine. Since, in many cases, the reason for passing by reference is to allow the subroutine to modify the value, it may be safe to assume that any parameter passed by reference will be modified. **But,** in the C language large variables such as strings and structures and strings are typically passed by reference to save space on the stack and for performance reasons – it takes a while to copy several hundred bytes onto the stack. In these cases, while it is *possible* for the subroutine to modify the variable, it may not happen.
High level languages such as C, Pascal, Java or Ada have their own ways to indicate whether a parameter is by value or by reference:

C: in C all parameters are by value, so you have to pass the address of the parameter using a pointer `*y` (and using the "address-of" operator, '&'):

```
int function1 (int x, int *y)
{
   x = 5;
   (*y) = 7;
}
```

The call `function1 (a, &b)` cannot modify the value of `a`, but it may modify the value of `b`. Please note that the actual parameters are `a` and the address of `b`.

Pascal: In Pascal, programmers use the word `var` to indicate an address parameter:

```
procedure1 (x: Integer; var y: Integer)
```

```
{
   x = 5;
   y = 7;
}
```

The call `procedure1 (a, b)` cannot modify the value of `a`, but it may modify the value of `b`.

Ada: In Ada there are actually 3 types of parameters: `in`, `out` and `in out`.

```
procedure1 (a: in Float; b: in out Float)
```

Parameters of mode `in` can only be read, parameters of mode `out` can only be written, and parameters of mode `in out` can be both read and written. From the implementation point of view, an `in` parameter is passed by value, an `in out` parameter is passed by address and initialised with the value of the actual parameter, and an `out` parameter is passed by address but not initialised.

12.1.1 Passing Parameters in Assembly

Consider the following assembly code segment:

Code Example 12.1: Subroutine argument passing

```
1          move.w A,-(sp)      ; push value of A on stack
2          pea    B            ; push reference to B on stack
           bsr    sub          ; branch to subroutine
4    sub   move.l 4(sp),a0     ; a0 points to B
           move.w (a0),d0       ; get B into d0
6          move.w 8(sp),d1     ; get A off the stack into d1
           .......
8          rts

10   data  org $2000
           dc.b  A  5           ; first variable
```

```
12          dc.b  B  7            ; second variable
```

The code pushes the value of A onto the stack, and the address of B onto the stack. The subroutine retrieves the values from the stack, copies then onto registers and operates with the copies. Hence, since we do not have the address of A on the stack, the original value of A will always stay the same, it cannot be changed from within the subroutine. In contrast, since we pushed the address of B onto the stack with the `pea` instruction, we are able to change the original value of B if we need to, like so:

```
move.b #9,(a0)      ; change the value of B to 9
```

In this way, a function or method may access the values of the arguments (*passed in*), and in some cases it may change the original value of a variable passed as an argument (in-out parameter).

Summarising, there are two mechanisms for passing parameters to a function or method:

By Value: The value of the variable is pushed on the stack.

- The actual value of the parameter is transferred to the subroutine/function/method.
- Unless the parameter needs to be updated, this is the safest approach because the original value cannot be changed by accident.
- In some HLLs it is possible to return a complex value (object/struct) to the caller, making it possible to change the value of a variable in the caller, as in:

```
my_record = update_record (my_record);
```

- It is not suitable for large amounts of data, since that would require a large copy onto the stack and perhaps back to memory.
- In order to pass a parameter by value through the stack, one may use the 68K instruction:

```
move.w A,-(sp)     ; A is pushed on the stack
```

By Reference: The address of the variable is pushed on the stack.

- The address of the parameter may be used by the subroutine/function/method.
- This will be necessary if the parameter is to be changed by the function.
- Recommended in the case of large data volume (object/struct/array), since only the address needs to be pushed on the stack, not the whole object/struct.
- In order to pass a parameter by reference through the stack, one may use the instruction:

```
pea B          ; address of B is pushed on the stack
```

The following example illustrates a calculation performed with parameters on the stack, in which the final result may be copied back to memory since the address of `reslt` has also been pushed on the stack.

Code Example 12.2: Program to sum squares using subroutine

```
1   ; data1^2 + data2^2 into reslt
2           org         $1000             ;main section
            move.l      #$7ffe,sp         ;set up the stack pointer
4           move.l      #reslt,-(sp)      ;pass addr. of reslt thru stack
            move.l      data1,-(sp)       ;pass data1 through stack
6           move.l      data2,-(sp)       ;pass data2 through stack
            bsr         sumsqr            ;call subroutine
8           lea         12(sp),sp         ;restores the stack pointer
            move.b      #9,d0             ; go back
10          trap        #15

12          org          $2000            ;subroutine
    sumqsr  lea          4(sp),a1         ;get starting address of params
14          move.l       (a1)+,d0         ;data1 to d0,
            move.l       (a1)+,d1         ;data2 to d1
16          move.l       (a1),a0          ;address of reslt goes to a0
```

```
            muls        d0,d0           ;d0 = d0 * d0
18          muls        d1,d1           ;d1 = d1 * d1
            add.l       d0,d1           ;d1 = d0 + d1
20          move.l      d1,(a0)         ;save result to reslt
            rts
22
            org         $3000           ;data section
24  data1   dc.l        $200
    data2   dc.l        $100
26  reslt   ds.l        1
            end
```

12.2 Stack Frames: (Activation Records)

The stack provides a convenient mechanism for dynamically allocating storage for data associated with the execution of a procedure. We have discussed how room is allocated on the stack when a procedure is called, and how that space must be deallocated when it returns. The stack should be left as it was before the procedure call. The block built on the stack to support a procedure call and its return is called the *stack frame* or *activation record*. The stack frame for a procedure is built by the compiler to contain all the information to be able to run the procedure, including space for:

- parameters
- local variables
- return value
- frame pointer `fp` (more on this below)

It would be only necessary for the calling program and the called procedure to agree on the structure of the stack frame for each procedure call. However, the specification of a standard calling convention makes possible the use of procedure libraries by defining the structure of the stack frame uniformly for all procedure calls. Compilers that follow

the calling convention generate code that will work correctly with procedures written in any high-level language. Figure 12.1 illustrates the structure of a stack frame[1]:

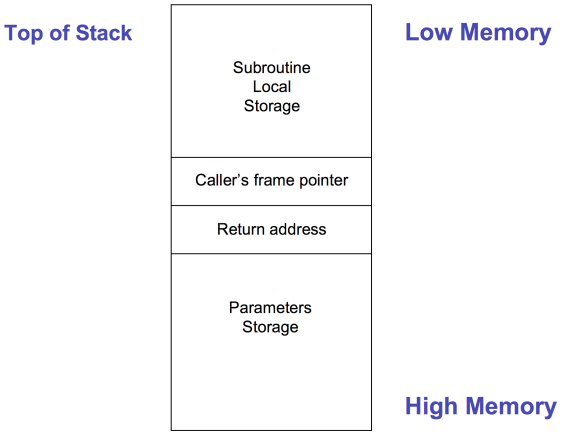

Figure 12.1: A standard stack frame

12.2.1 The Frame Pointer

We have just introduced a new pointer `fp`, relevant to the execution of a procedure. In previous examples we have shown how to use the stack pointer `sp` to `push()` and `pop()` elements, and to calculate the location of the parameters used by the procedure. However, this presents a problem, since the `sp` can change during the execution of a procedure — for example when the procedure dynamically allocates a block of storage on the stack — so data on the stack frame cannot reliably be referenced through offsets from the `sp`. Thus, the stack pointer can be used only immediately after the invocation of the procedure, since the `sp` is not guaranteed to have the same value throughout the execution of the procedure.

Instead, the `fp` is set to a fixed value within the stack frame, typically pointing just above the return address as shown in Figure 12.1. The frame pointer remains fixed, that is, the `fp` is guaranteed to have the same value throughout the execution of the procedure, and therefore all local data can be accessed via fixed offsets from the `fp`.

[1]The meaning of the label 'Caller's stack frame' will be apparent in the next section.

Then offsets from `fp` will always be correct, while offsets from `sp` will depend on the allocations of the dynamic area.

In the 68K `a7` is the `sp`, and by convention `a6` is typically used as the `fp`, although `a6` has no special properties. Consequently, if we follow these conventions we can use the value in `a6` to calculate all the necessary offsets to run the procedure. Figure 12.2 shows the two registers holding the values of the respective pointers.

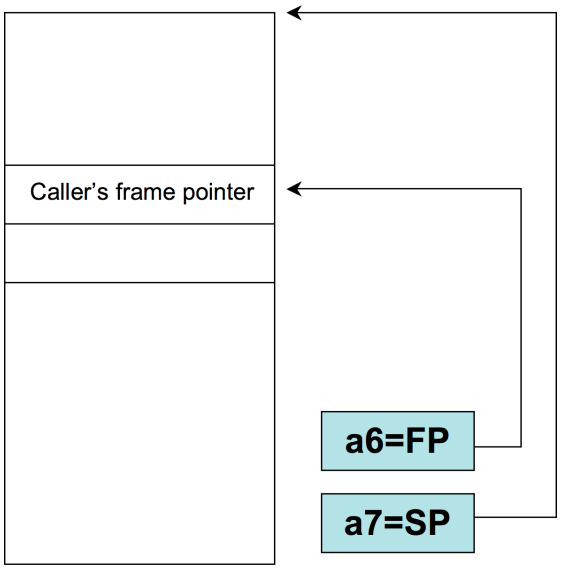

Figure 12.2: A standard stack frame and its pointers

12.2.2 Nested Subroutine Calls

We have seen that a subroutine may call another one while executing. In that case, the stack frame for the called routine is naturally built on top of the previous one. However, since we are using `a6` as the `fp`, we have to change the value of `a6` to point to the new invoked routine. Since we don't want to lose the value of the caller's `fp`, we store in the location pointed to by `a6` the address of the caller's `fp`, as depicted in Figure 12.3.

Please note that the *address* of the `fp` is used to calculate the offsets for the running

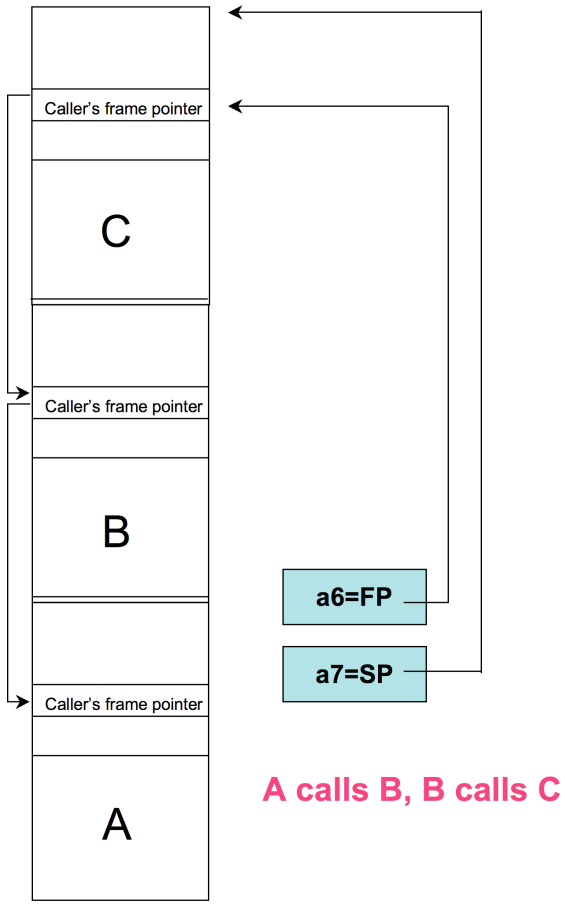

Figure 12.3: Nested stack frames

routine, while the *contents* of that location is used to point to the caller's `fp`. This:

- allows a called procedure to access values (variables, parameters, etc.) of the caller, and of the caller's caller, etc., by following the chain of pointers;
- importantly, since a caller is suspended while an invocation is proceeding, the caller cannot access the data in the invoked procedure; this is because the data is not there when the call is made, and then it is deleted when the caller returns;
- makes possible to implement an arbitrary number of such calls, since there is no limit to the number of frames that can be built on top of each other (except the limit of available memory).

Note: in particular, a routine may call itself repeatedly: this is called *recursion*.

12.2.3 `link` and `unlk`

The 68K provides two special instructions to allocate and deallocate a stack frame. The `link` instruction takes an argument register — usually assumed to be `a6` — and an immediate constant. It is intended to be the first instruction executed by a called routine. If `an` is used as the argument pointer, the instruction:

```
link an,d      ; an is assumed to be a6
```

1. Pushes `an` on the stack, that is, it copies `an` at `-(sp)`.
2. Puts `sp` into `an`, so the current stack pointer `sp` is pointed to by the frame pointer register, `an`.
3. Changes `sp` to `sp-d`, i.e. decrements the stack pointer `sp` by `d` (`d` is assumed to be the space necessary for local variables, see below).

This is exactly what is required by the call. Let us go through the sequence:

- just before a routine is called the parameters are pushed on the stack, thus building the lower part of the activation record shown in Figure 12.1;
- the call to the routine pushes the return address on the stack, on top of the parameter space, as indicated in Figure 12.1;

- the link instruction pushes the current `an` on the stack, because this is pointing to the caller's activation record, as shown in Figures 12.1 and 12.2;
- a space of size `d` is allocated on the stack to be used by local variables.

Conversely, the instruction `unlk` is intended to be the last instruction executed by the routine, just before the `rts` instruction.

```
unlk an      ; an is assumed to be the frame pointer
```

The address register `an` is assumed to be the frame pointer. The `unlk` instruction replaces the value of the stack pointer `sp` with the contents of `an`, and then pops the stack into `an` (i.e. loads into `an` the value `(sp)+`). That is:

1. `an` → `sp`, change the value of the `sp` to the contents of `an`, so the stack pointer and the frame pointer have the same value, in effect deallocating the local variables apace.
2. `(sp)+` → `an`, put the value stored on the stack (the caller's frame pointer) into `an` and deallocate that stack space, so the stack pointer now points just above the return address ready for the `rts` instruction.

After the return, the routine has left the stack as it found it (the golden rule), only the parameters to the call remain on the stack. However, as we have discussed before, it is responsibility of the caller to remove these.

Due to the location of the `fp` on the stack, return addresses and passed arguments are always positive relative to the frame pointer `fp`, as the following examples illustrate.

Example 1: A simple use of the `link-unlk`:

```
        move.w    d0,-(sp)    ;push parameter #1 onto stack
        move.w    d1,-(sp)    ;push parameter #2 onto stack
        jsr       sbrt        ;jump to subroutine sbrt

sbrt    link      a6,#$-8     ;establish fp and local storage
```

```
. . .
move.w    10(a6),d5   ;retrieve parameter #1 using fp
. . .
unlk      a6          ;fp for the calling routine re-established
rts                   ;deallocate stack frame and return
```

Example 2: A more complex example:

```
n       equ     8               ;8 bytes for output
m       equ     8               ;8 bytes for local variables

        add.l   #-n,sp          ;put output area on stack
        move.l  arg,-(sp)       ;put argument on stack
        pea     x               ;put address of data table on stack
        jsr     subr            ;goto subroutine
        add     #8,sp
        move.l  (sp)+,d1        ;read outputs
        move.l  (sp)+,d2
        . . .

subr    link    a1,#-m          ;save old sp
        . . .
        move.l  local1,-4(a1)   ;save old variables
        move.l  local2,-8(a1)
        . . .
        add.l   #1,-4(a1)       ;change a local variable
        movea.l 8(a1),a2        ;get x
        . . .
        move.l  output,16(a1)   ;push an output
        . . .
        unlk    a1
        rts

local1  dc.l    $98765432       ;local variables
local2  dc.l    $87654321
output  dc.l    'adcb'          ;output value
```

Example 3: Example of a function call for the 68K:

```
;  func(a, b); a and b parameters
;  a6 is the frame pointer and a7 the stack pointer,

       move        a, -(a7)          ; push a on the stack, a7
       move        b, -(a7)          ; push b
       jsr         func              ; push return adress on the stack
                                     ; and jump to func
;  int func(int a, int b)
       link        a6,#-8            ; push a6 on the stack, move a7
                                     ; into a6. a6 is now the frame pointer
                                     ; of the called function. decrement a7
                                     ; by 8 to allocate storage for a and b
                                     ; a6 is the frame pointer and is used to
                                     ; reference local variables
                                     ; a7 is the stack pointer and is used to
                                     ; store return addresses for later
                                     ; function calls
...
   unlk a6                           ; undo link: copy a6 into a7,
                                     ; retrieve old a6 from the stack
   rts                               ; pop return address from the stack
```

12.3 Variable Scope and Visibility

From the previous section, we can see that a sequence of procedure calls results in a collection of stack frames piling up, as shown in Figure 12.3. Note in the figure that when the last procedure call is executing, all the other caller procedures have stopped executing. In other words, when the last procedure `procedure3()` is executing, the second last — `procedure2()` — is waiting for it to return to continue running. But since `procedure3()` stack frame is wiped out when it returns, `procedure2()` has no chance to be able to use any of the data on `procedure3()` stack frame. However, the stack frame for `procedure2()` is available when `procedure3()` is executing, and the data is accessible following the pointer from `procedure3()` to `procedure2()` frame pointer, so `procedure3()` is able to access data on `procedure2()` stack frame. Naturally, the same applies for `procedure1()`, so `procedure3()` can access data in `procedure1()` stack frame.

This characteristic relates to *variable scope and visibility*, because the definition of a variable is restricted to the block where it has been defined, and local variables of an inner block are 'invisible' to outer blocks. The rule is as follows:

- a calling procedure is unable to access variables or other data local to the called procedure;
- a called procedure is able to access variables or other data local to the calling procedure.

Practical Work 12

12.4 Tutorial exercises

1. Type and run the following sample programs. Explain what the programs do, and make sure you can describe all the instructions and their behaviour. The EASy68K simulator is a bit too helpful and automatically echos characters input. Lines 8-10 of the following code turn off the echo so we can do it ourselves.

Code Example 12.3: Read a string

```
2   ; reads characters from the keyboard into a string
    ; the chars are echoed on screen, loops until return
4
    start org      $1000          ; start
6         move.l   #$7ffe,sp      ; initialise stack
          move     #$2000,a1      ; address of string in a1
8         move.b   #0,d1          ; turn off automatic keyboard echo
          move.b   #12,d0
10        trap     #15

12  nxt   move.b   #5,d0          ; INCH
          trap     #15
14        move.b   #6,d0          ; OUTCH
          trap     #15
16        move.b   d1,(a1)+       ; store char
          cmp.b    #$0d,d1        ; is it <return>?
18        beq      ext            ; exit if it is
          bra      nxt            ; otherwise next
20  ext   move.b   #9,d0          ; back to simulator
          trap     #15
22
          org      $2000          ; start of string
24        ds.b     256            ; room for the chars
          end      start
```

Code Example 12.4: Convert case

```
; It converts lowercase chars to uppercase as they are entered.
; Other chars are left unchanged.  The chars are not stored
; they are converted and displayed until the user hits return

        org         $1000         ; start
        move.l      #$7ffe,sp     ; initialise stack
        move.b      #0,d1         ; turn off automatic keyboard echo
        move.b      #12,d0
        trap        #15

nxt     move.b      #5,d0       ; INCH
        trap        #15
        cmp.b       #$0d,d1       ; is it <return>?
        beq         ext           ; exit if it is

        cmpi.b      #'a',d1       ; if <'a' not lowercase
        blt         display       ; so display
        cmpi.b      #'z',d1       ; if >'z' not lowercase
        bgt         display       ; so display

        subi.b      #32,d1        ; to upper

display move.b      #6,d0       ; OUTCH
        trap        #15
        bra         nxt           ; otherwise next
ext     move.b      #9,d0       ; go back
        trap        #15
        end
```

2. Explain how subroutines (methods/functions) in languages such as Java and C can both accept parameters *and* return values to the method that called them. Do so with a diagram showing the stack, and some sample code that demonstrates the subroutine returning the value and the value being retrieved by the original caller. How is a return value implemented in assembly using the stack?

3. Write an assembly program which accepts up to *100* characters from the user and echoes them to the screen. The entered string is terminated by the newline character. The program should count the number of vowels, consonants and non-letter characters. Each of these counts should be displayed in hexadecimal on the screen before the program exits.

   - What subroutines would be useful for this program?
   - Write the program using these subroutines with parameter passing where appropriate.

12.5 Laboratory exercises

Use this time to work on Assignment 3. Consult your lab assistant with any problems you may be experiencing.